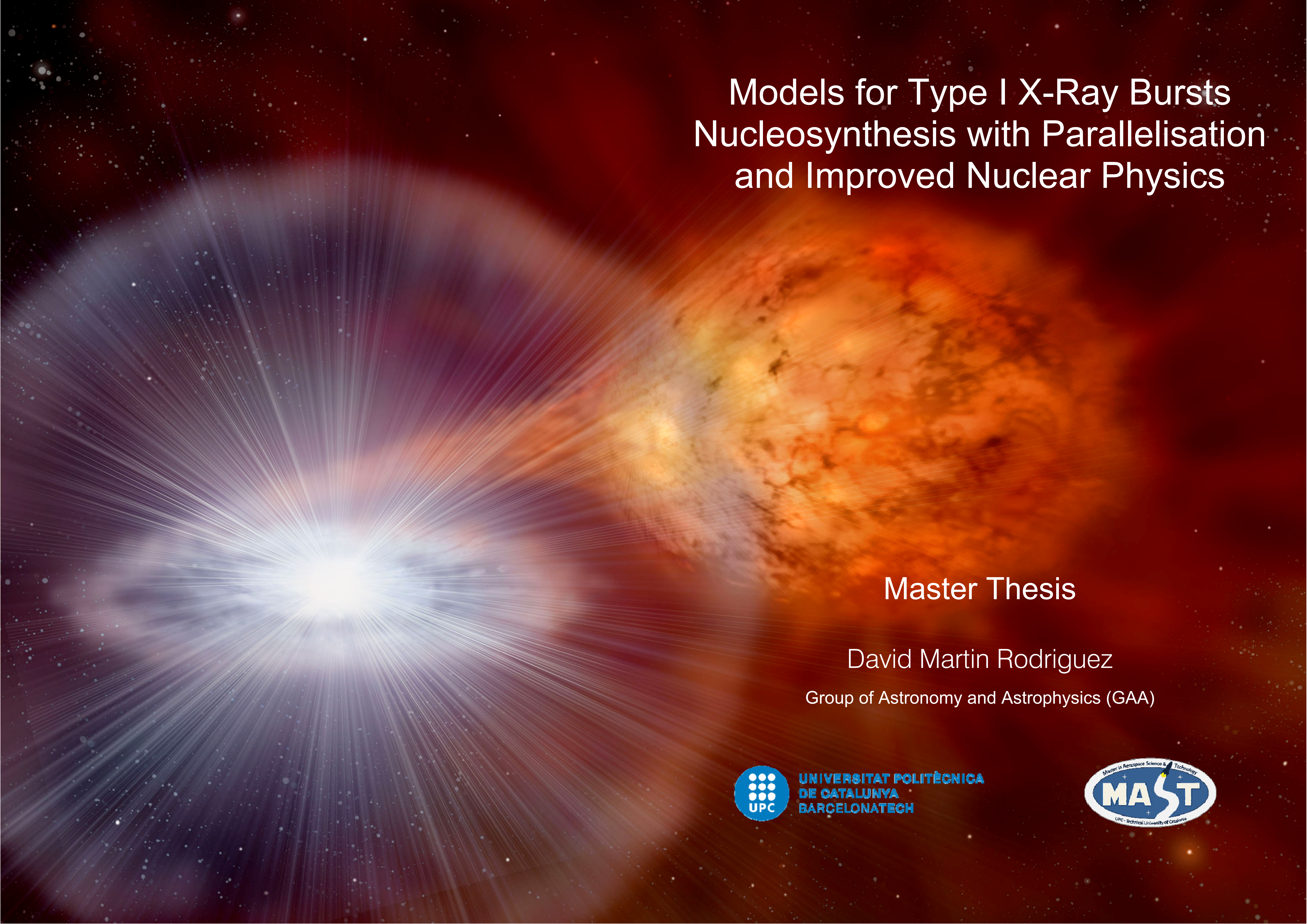


Models for Type I X-Ray Bursts Nucleosynthesis with Parallelisation and Improved Nuclear Physics

Master Thesis

David Martin Rodriguez

Group of Astronomy and Astrophysics (GAA)

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

MAST

Master in Aerospace Science & Technology

UPC - Technical University of Catalonia

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

MASTER THESIS

Models for Type I X-Ray Bursts Nucleosynthesis with Parallelisation and Improved Nuclear Physics

David Martin Rodriguez

SUPERVISED BY

Jordi José

DEPARTAMENT DE FÍSICA I ENGINYERIA NUCLEAR

Universitat Politècnica de Catalunya
Master in Aerospace Science & Technology
February 2012

This Page Intentionally Left Blank

Models for Type I X-Ray Bursts Nucleosynthesis with Parallelisation and Improved Nuclear Physics

BY

David Martin Rodriguez

DIPLOMA THESIS FOR DEGREE

Master in Aerospace Science and Technology

AT

Universitat Politècnica de Catalunya

SUPERVISED BY:

Jordi José

DEPARTAMENT DE FÍSICA I ENGINYERIA NUCLEAR

This Page Intentionally Left Blank

A mi familia:
a la que ya no está,
a la que ha de venir.

This Page Intentionally Left Blank

ABSTRACT

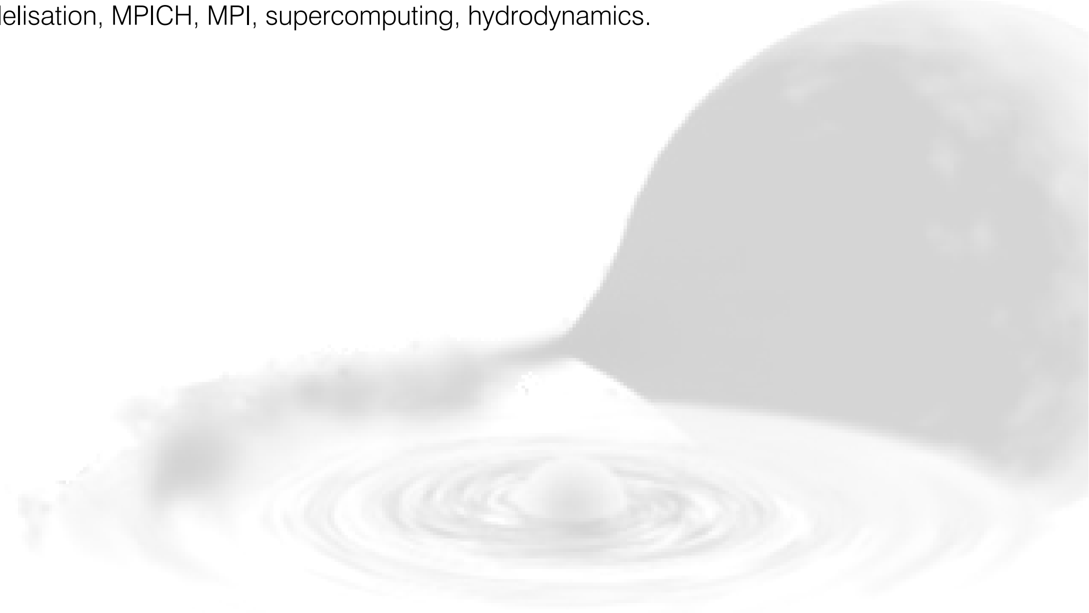
Type I XRBs are thermonuclear flashes on the surface of neutron stars (NS) associated with mass-accretion from a companion star. Models of type I XRBs and their associated nucleosynthesis are physically complicated and extremely intense as regards the huge computational power required to model the physical processes played out, with the required precision to be truly representative. Until recently, because of these computational limitations, studies of XRB nucleosynthesis have been performed using limited nuclear reaction networks. In the bid to overcome this hurdle, parallel computing has been raised as the main permitting factor of yet more precise and computationally intensive simulations as it offers the potential to concentrate computational resources on intensive computational problems. In this Work, we present a parallelisation of two different applications; a one-zone (i.e. parameterized) nucleosynthesis code, and a one-dimensional (spherically symmetric), hydrodynamic code, in Lagrangian formulation (hereafter SHIVA code), built originally to model classical nova outbursts (José 1996; José & Hernanz 1998).

The codes have been parallelised using the MPICH2 implementation of the Message Passing Interface (MPI) specification for the design of parallel applications using clusters of distributed workstations. As an example, to execute a hydrodynamic simulation along 200k time-steps, the SHIVA code requires (in its sequential, single-node version) about 147 hours (6.1 days) to complete when using a reduced nuclear network with 324 isotopes and 1392 nuclear reactions, and 688 hours (28.6 days) when using a network with 606 nuclides and 3551 nuclear reactions for the same number of time-steps.

The post-processing nucleosynthesis code is a time-step loosely synchronous application with a very small problem size (limited by the number of isotopes of the nuclear network). As shown by the performance tests, this fact results in the worst possible scenario for parallelisation; results show that the performance of the parallel application is much worst than the sequential, 1-node version of the code. Our results show that it is therefore not possible to parallelise efficiently a post-processing nucleosynthesis code, and efforts in this regard should be avoided. On the contrary, the parallelised version of the SHIVA code yields excellent performance results. A speed-up factor of 26 is achieved in a simulation with a reduced network consisting of 324 isotopes and 1392 nuclear reactions when 42 processors are used in parallel to execute the application along 200k time-steps. On the other hand, an excellent speed-up factor of 35 is accomplished in a simulation with a reaction network up to 606 nuclides and 3551 nuclear reactions. Maximum speed-ups of $\sim$41 and $\sim$85 are predicted by the performance models when using 200 processors, for the reduced and extended simulations respectively.

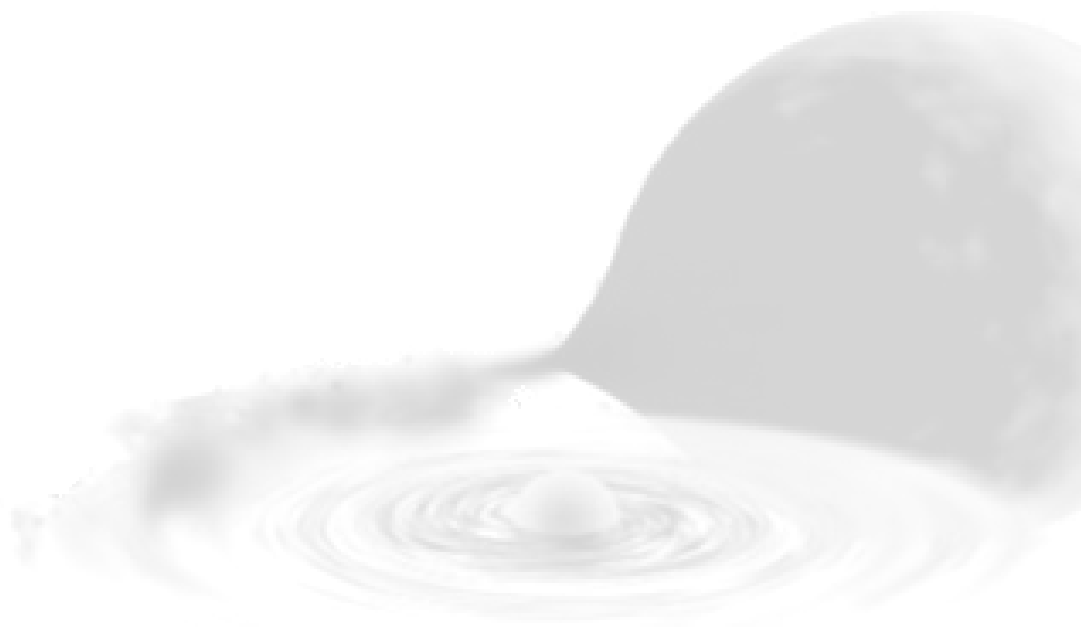
Our results will not only improve the quality of the simulations (and hence publications) in terms of better numerical approaches, finer approximations, and a considerably shorter time-to-publication, but also will allow taking advantage, if desired, of parallel supercomputing facilities like the *Mare Nostrum* at the Supercomputing Centre in Barcelona (BSC).

Subject headings: nuclear reactions, nucleosynthesis, abundances — stars: neutron — X-rays: bursts, parallelisation, MPICH, MPI, supercomputing, hydrodynamics.



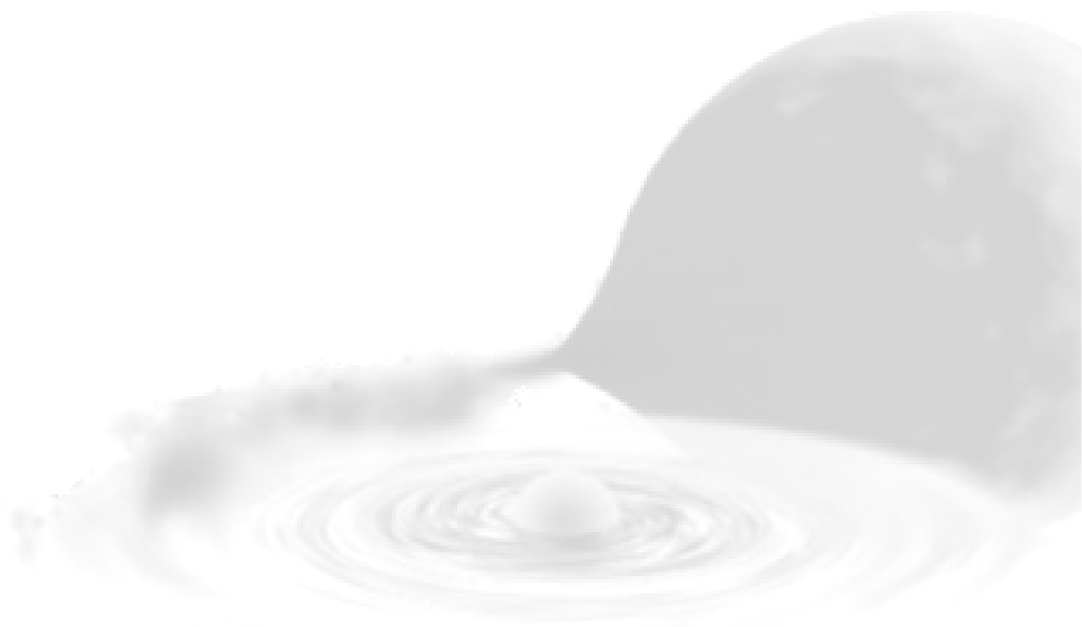
This Page Intentionally Left Blank

Contents



List of Figures

List of Tables



INTRODUCTION

Type I X-ray bursts (XRBs) are cataclysmic stellar events powered by thermonuclear runaways (TNRs) in the H/He-rich envelopes accreted onto neutron stars in close binary systems. They constitute the most frequent type of thermonuclear stellar explosion in the Galaxy (the third, in terms of total energy output after supernovae and classical novae) because of their short recurrence period (hours to days). More than 90 Galactic low-mass X-ray binaries exhibiting such bursting behaviour (with typical durations of $10-100\,s$) have been found since the discovery of XRBs in 1976 (Grindlay *et al*. 1976, Belian *et al*. 1976).

Modelling of type I XRBs and their associated nucleosynthesis has been extensively addressed by different groups, reflecting the astrophysical interest in determining the nuclear processes that power the explosion, as well as in providing reliable estimates for the chemical composition of the neutron star surface. Indeed, several thermal, radiative, electrical, and mechanical properties of the neutron star depend critically on the specific chemical abundance pattern of its outer layers.

Models of type I XRBs and their associated nucleosynthesis are physically complicated and extremely intense as regards the huge computational power required to model the abovementioned physical processes with the required precision to be truly representative. Until recently, because of these computational limitations, studies of XRB nucleosynthesis have been performed using limited nuclear reaction networks, truncated around Ni, Kr, Cd, or Y. On the other hand, Schatz *et al*. (1999, 2001a) have carried out very detailed nucleosynthesis calculations with a network containing more than 600 isotopes (up to Xe, in Schatz *et al*. 2001a), but using a one-zone approach. Koike *et al*. (2004) have also performed detailed one-zone nucleosynthesis calculations, with temperature and density profiles obtained from a spherically symmetric evolutionary code, linked to a 1270-isotope network extending up to ^{198}Bi.

The simulation of accurate models is an especially high time-consuming task. Computational cost of these simulations typically increase as the fourth power or more of the 'resolution' that determines accuracy, so these studies have a seemingly insatiable demand for more computer power. They are also often characterised by large memory and input/output requirements. For example a post-processing code that runs about 50,000 post-processing calculations, with a network containing 606 nuclides (H to ^{113}Xe) and more than 3500 nuclear processes, requires about 9.1 CPU-months of calculating power to compute 1 burst using a model with 200 shells (Moreno 2009).

In this regard, parallel computing has been raised as the main permitting factor of more and more precise, computationally intensive simulations. A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations, multiple-processor workstations, and embedded systems. Parallel computers are interesting because they offer the potential to concentrate computational resources---whether processors, memory, or I/O bandwidth---on important computational problems.

In the first part of this Work, a type I XRBs post-processing nucleosynthesis code has been parallelised using the MPICH2 implementation of the Message Passing Interface (MPI) specification for the design of parallel applications using clusters of distributed workstations. In the second part, we have successfully parallelised the spherically symmetric, Lagrangian, hydrodynamic code SHIVA (José 1996; José & Hernanz 1998), in pursuit of significant speed-ups that allow for detailed hydrodynamic simulations with extended nuclear reaction networks in affordable times. This problem architecture represents the so called fully synchronous parallelism (section 2.3.2), indicating that each one of the computations is performed synchronously (or simultaneously) to all data. The main point here is that all future calculations of decisions hinge on the results of the earlier, preceding data calculations. Parallelisation can be achieved by having each node actually cycling through a subset of the neutron star envelope shells (i.e. a number of contiguous shells).

Parallelism is an enticing and intuitive concept. In principle, parallelism is as simple as applying several processors or CPUs to a single problem, so that if it takes, say, one hundred hours to complete, we may put ten CPUs to work on the problem and get the results in just ten hours. In practice, however, parallelisation takes a high toll on both engineering and programming efforts, as it will be shown shortly. On top of that, depending on the nature of the problem being parallelised it may turn out that parallelisation does not pay off altogether. Be that as it may, if the problem at hand fits into the parallelisable categories, then in addition to offering faster solutions, codes that are converted to parallel, are capable of solving larger, more complex problems. Simulations can be executed at a much finer resolution, and also physical phenomena may be modelled far more realistically.

One of the purposes of the parallelisation is to benefit from the 42-node Hyperion Cluster that the Astronomy and Astrophysics Group (GAA) has at the EUETIB (UPC), and speed-up considerably the execution of the code. This will allow for better and more accurate simulations (e.g. with more isotopes and reactions, or with a significant increase in the resolution, in terms of more layers of the neutron star envelope's model), in affordable execution times. This will not only improve the quality of the simulations (and hence publications) in terms of better numerical approaches and approximations, but will also allow to take advantage of parallel supercomputing facilities.

An outline of this Master Thesis is as follows: Chapter 1 describes stellar binary systems and X-ray binaries in particular. Also it is provided a brief description of neutron stars, and why its understanding is important and how X-ray bursts can be used in determining the properties of the neutron star. Chapter 2 provides an introduction to parallel computing, and the type of applications that can be parallelised. Emphasis is placed on analysing the properties of an application that make for better and more efficient parallelisation. It is also described the 42-node Cluster Hyperion. Chapter 3 presents the parallelisation of the post-processing nucleosynthesis application. It is shown the strategy followed to parallelise it as well as the tools used. Performance results are summarised and discussed. Chapter 4 describes the parallelisation strategy of the fully coupled hydrodynamic code SHIVA. A model to predict the performance of the parallelisation is included, and performance results are shown and discussed. Finally, Chapter 5 summarises the main points, results, and achievements of this Thesis, and provides several points that are open for further research and improvement.

CHAPTER 1
THE PHYSICS CASE

"If the hand be held between the discharge-tube and the screen, the darker shadow of the bones is seen within the slightly dark shadow-image of the hand itself... For brevity's sake I shall use the expression 'rays'; and to distinguish them from others of this name I shall call them 'X-rays'."

Wilhelm Conrad Röntgen
'On a New Kind of Rays' (1895). In Herbert S. Klickstein, Wilhelm Conrad Rontgen: On a New Kind of Rays, A Bibliographic Study (1966), 4.

1.1. Binary Star Systems

Binary star systems contain two stars that orbit around their common centre of mass. Many of the stars in our Galaxy are part of a binary system. According to statistics, it is conceivable that approximately one half of all stars may belong to a binary system. The two stars belonging to a binary system have a significant influence on each other's evolution. In these systems, the distance between the two stars may range from a few times the radii of the stars to a completely different situation where there is an envelope common to both stars (this type of binary systems are called contact binaries). Consider the binary star system shown in Fig. 1 below. Each of the stars is surrounded by an equipotential tear-shaped volume. If and where the two equipotentials of the stars touch, they are designated with the term *Roche lobe*. In the figure below, it is shown as a dashed curve with the shape of an '8', its intersection with the equatorial plane.

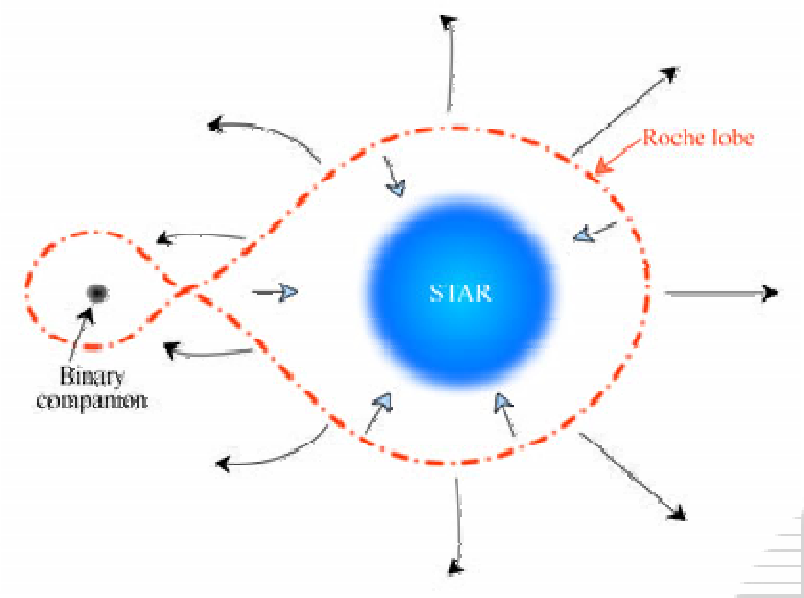


Fig. 1 Binary star system with Roche lobe representation (The SAO Encyclopaedia of Astronomy)

There is a point where the effects of rotation and gravity cancel each other. This special point is called the *inner Lagrangian point*. It may happen that the companion star fills its Roche lobe (Fig. 2). This could occur for instance when one of the stars becomes a red giant, or even for main sequence stars if they are close enough to their companion star. In this situation, matter is then accreted from that star through the inner Lagrangian point onto the surface of the companion.

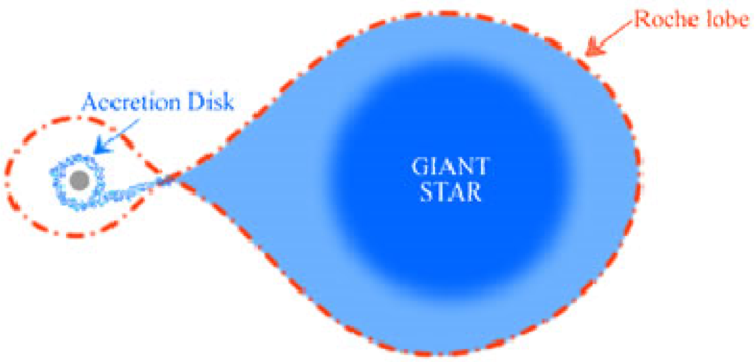


Fig. 2 Binary star system with accretion material (The SAO Encyclopaedia of Astronomy)

There is a special class of binary stars called X-ray binaries. They are so-called because they predominantly emit in X-rays. X-ray binaries are made up of a normal star and a collapsed star (a white dwarf, neutron star, or black hole). These pairs of stars produce X-rays if the stars are close enough together that material is accreted from the normal star by the gravity of the more dense, collapsed star, by means of Roche lobe overflow. The X-rays come from the area around the collapsed star where the material that is falling toward it is heated to extremely high temperatures.

1.2. X-ray Binaries

Many of the close binary star systems contain a neutron star as a compact object. A neutron star could have a mass of approximately 1.4 $M_\odot$ densely packed within a radius between 10 and 15 km. The central density of the neutron star may reach 10^{15} g/cm^3 (see Lattimer and Prakash 2004). An enormous gravitational energy release is produced when the matter from the companion star is accreted on the surface of the neutron star. This provokes that the temperatures at the neutron star surface soar to approximately 10^7-10^8 K, while there is a persistent emission that occurs predominantly at X-ray energies (Lewin *et al.* 1993).

X-ray binaries may be further classified as high-mass binaries and low-mass binaries. For high-mass binaries, the companion star is a massive population I star ($\geq$ 5 $M_\odot$), whereas the compact neutron star is highly magnetised. This provokes that the accreted matter is transferred at high velocities and is channelled along the lines of the magnetic field onto the magnetic poles. Consequently a hot spot of X-ray emission is created and, if the axis of rotation of the neutron star is inclined with respect to the magnetic axis, it generates an X-ray pulsar. There are typical rotation periods from 0.1 seconds up to a fraction of an hour. It has been observed that the periods of rotation of some X-ray sources decrease, which may be an indication that the neutron star is spun up as a consequence of the accretion of matter from the companion star.

On the other hand, low-mass X-ray binaries exist where the companion is a low mass star ($\leq$ 1.5 $M_\odot$) and where the neutron star has a weak magnetic field. In this scenario matter is transferred via Roche lobe overflow. In a fair number of cases these systems produce bursts in the X-ray frequencies, as well as a persistent X-ray emission (Lewin, van Paradijs and Taam 1993). Additionally, it has been observed a rare kind, called type II X-ray bursts, where the bursts occur in rapid succession with a separation of just a few minutes between them. There is an abrupt rise and fall of the profile of each of these bursts. They are supposed to be associated with a rapid increase in the mass accretion rate provoked by instabilities in the accretion disk.

1.3. Type I X-ray Bursts

Most of the bursts can be classified as type I X-ray bursts where increases of an order of magnitude are seen in the X-ray luminosity. Type I XRBs are thermonuclear flashes on the H/He-rich envelopes of accreted matter onto neutron stars (NS). The hydrogen- and helium-rich matter pulled out from the companion star has high angular momentum; therefore it does not settle down directly but forms a rapidly rotating accretion disk. As accreted matter losses momentum, it settles down onto the neutron star surface. The temperatures accomplished in the accreted matter because of compression heating are high enough to maintain a continuous fusion of hydrogen into helium via the hot CNO cycles (Bildsten *et al.* 2007, Iliadis 2007). As the temperature rises, helium is finally ignited via the triple-α reaction, which makes the temperature soar and as a consequence a thermonuclear runaway occurs. Some nucleosynthesis models predicts that the mixture of hydrogen and helium of the outer region will afterwards be burnt explosively, whereas other models indicate that the ignition may occur in pure helium or in mixed hydrogen-helium material. Depending on the specific temperatures and densities reached throughout the process, different nucleosynthesis phenomena may be obtained in the several burning layers of the surface of the neutron star. Computations have revealed that in the deepest, hottest and densest layers, elements beyond the iron peak are synthesised (Woosley *et al* 2004, José *et al.* 2011, Fisker *et al.* 2006). At the end, the ashes of the burst form a new shell of matter onto which new material is again accreted from the companion star, and the cycle repeats.

The model shown in Fig. 3 explains the basic features of type I X-ray bursts. A burst lasts typically for less than 1 min and repeats after several hours to days. The luminosity profile shows a rapid rise within $\approx$1-10s, caused by the sudden nuclear energy release, and a slower decline on the order of $\approx$5-100s, reflecting the cooling of the neutron star surface. Some bursts show millisecond oscillations of the X-ray flux. These have been suggested to arise form a surface wave in the nuclear burning layer or perhaps from anisotropies in the nuclear burning caused by a spreading hot spot on the surface of a rapidly spinning neutron star.

To date, 96 Galactic low-mass X-ray binaries exhibiting such bursting behaviour (with $\tau_{burst} \sim 10-100s$) have been found since the discovery of XRBs in 1976 (Grindlay *et al* 1976), and (Belian *et al* 1976). A list of the current 96 Galactic Type-I X-ray bursters can be found in http://www.sron.nl/~jeanz/bursterlist.html. X-ray bursts were detected as bright sources in the X-ray band of the electromagnetic spectrum, which can only be observed out of the Earth's atmosphere. Grindlay *et al.* (1976) observed two bursts from a previously known X-ray source in the globular cluster NGC 6624. As of that initial, pioneer detection, multiple additional sources have been detected and reported; Lewin *et al.* (1993), Liu *et al.* (2007), and in 't Zand *et al.* (2009).

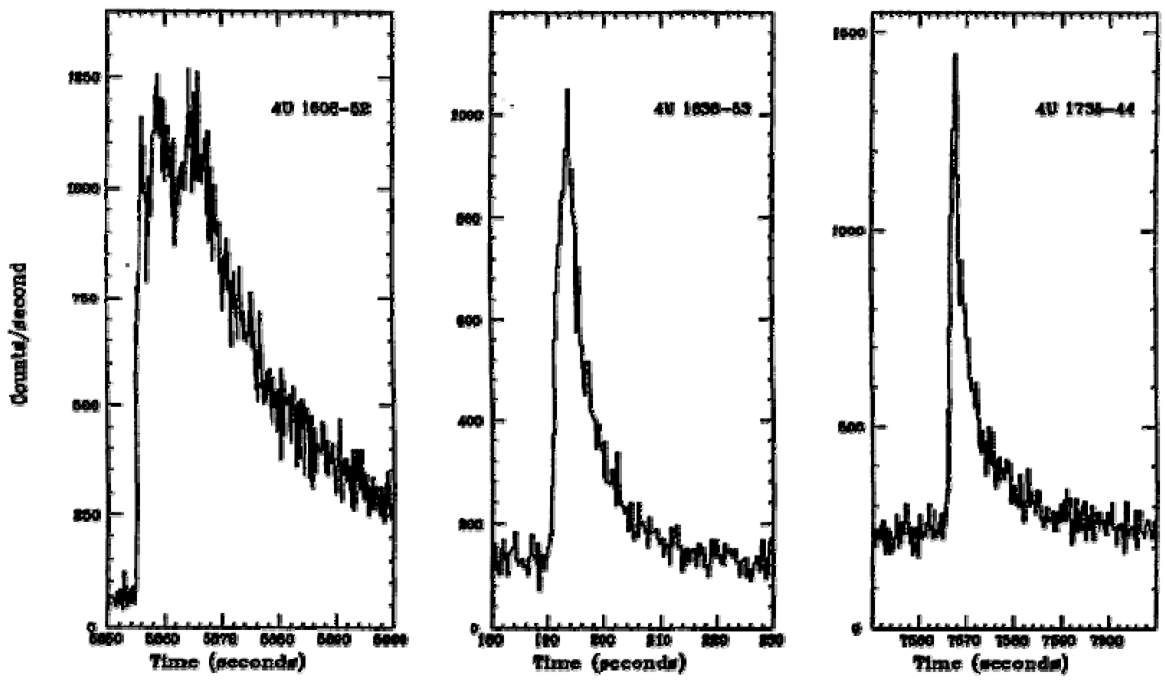


Fig. 3 Examples of X-Ray Bursts Profiles[1]

A number of different assumptions and parameters have a direct effect on the models of type I X-ray bursts. Examples of these parameters are the mass accretion rate, the number of points of ignition, the rotation velocity, how the burning front is propagated across the surface of the neutron star, and finally the composition of the accreted matter.

According to Woosley *et al.* (2004), in order to simulate realistically Type I X-ray bursts, researchers must cope first with four main difficulties. Firstly, it has to be known the geometry of the runaway as well as the physics underneath the accretion process. Secondly, there is a high complexity in the nuclear physics, where the interplay of the several reactions on the final result is still under investigation and where there is no single reaction or group of reactions that, on their own, drive the energy generation rate. The simulation must include all recent advances in that regard, like recent progress in the understanding of the major flows in the *rp*- and αp-processes and the properties of the isotopes involved (e.g., Schatz *et al.* 2001a, 2001b; Brown *et al.* 2002). The third difficulty is accomplishing the simulation of not only one burst but a number of successive bursts. As it was already indicated by Taam (1980) the *inertia* of the neutron star in terms of the evolution of its temperature and composition may have important implications on the properties of subsequent bursts. Only if a succession of X-ray bursts are simulated, the temperature effects and changes in the composition due to previous bursts can be taken into account (Ayasli & Joss 1982; Woosley & Weaver 1984).

[1] Lewin *et al.* (1993). Examples of X-Ray burst profiles as observed with EXOSAT in the ~1-20 keV energy band. The counting rates have not been corrected for dead time and refer to one half of the detector array. The horizontal axes represent time in seconds.

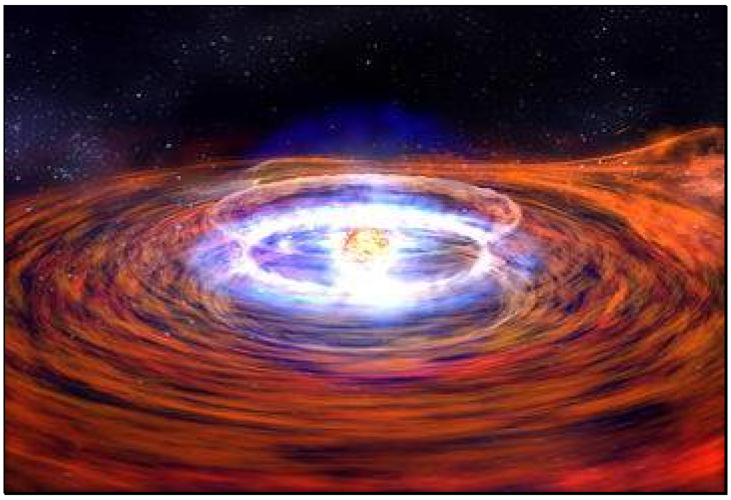


Fig. 4 Artist's concept of a neutron star X-ray burst (Credit: NASA/Dana Berr)

The fourth challenge is have access to as many photometry and spectra data as possible. There is a rich archive of photometry for X-ray bursts observations that must be taken into account in the models. Some models are limited to single-temperature black bodies, which are usually calculated using flux-limited radiative diffusion. Detailed studies of the colour temperature and spectrum may make use of these results as input to a more sophisticated treatment of the radiation transport.

All in all, it is only with a better understanding and modelling of the mechanisms governing the explosive X-ray bursts, that it is possible to determine the final abundance distribution following a burst, and therefore the abundance distribution that falls back onto the neutron star crust. This information is vital for the correct interpretation of the astronomical observations of neutron stars and to determine the physics occurring on their surfaces.

1.4. Understanding Neutron Stars

During the terminal phases of the evolution of a massive star ($M \geq 10\ M_\odot$), part of the mass of the star is lost in an explosion. There is a possibility that the remnant mass falls between 1.4 $M_\odot$ and 2 $M_\odot$, in which case the star collapses into a neutron star; a compact object with a radius of about 10 km to 15 km, but with dense core where the protons and electrons have merged into neutrons (Fig. 5). The average central density of a neutron star is 10^{15} g/cm^3 and the gravity on the surface would be around 10^{12} m/s^2 (Lattimer and Prakash, 2004).

Bildsten and Strohmayer (1999) present a review of current research concerning neutron stars, in which the authors make the following points:

1. With a density comparable to that of an atomic nucleus, a neutron star provides an extreme environment for fast and violent phenomena. Matter orbiting a neutron star can have a period as short as a millisecond. When such matter crashes into the star (i.e., is "accreted" by the star), such matter can be moving at one-third the speed of light. In

general, because their behaviour can vary over readily observable timescales, neutron stars can be rich sources of information about nuclear physics, general relativity, and astrophysics.

2. Though relatively elusive, neutron stars have been detected and studied over a broad range of electromagnetic frequencies, from radio frequencies to gamma rays. To date, astronomers have identified more than 1000 of the estimated 10^8 neutron stars in our galaxy. New orbiting astronomical satellites have produced recent rapid growth in our knowledge of these objects, with much of the progress occurring in our understanding of neutron stars that undergo sudden large energy releases.
3. The precise timing of radio pulsars has yielded astonishing astronomical discoveries, such as multiple Earth-mass planets orbiting a neutron star, and the direct confirmation of the loss of orbital angular momentum due to gravitational radiation in a double neutron star binary system (for which Russell Hulse and Joseph Taylor received the 1993 Nobel Prize in Physics). The brightest accreting neutron stars reside in binary systems and accrete matter from their companions (see section 1.2). These accreting neutron stars typically have luminosities more than a thousand times that of the Sun.
4. There is every reason to believe that new classes of neutron stars will be discovered by continued observations from the currently orbiting x-ray and gamma-ray satellites.

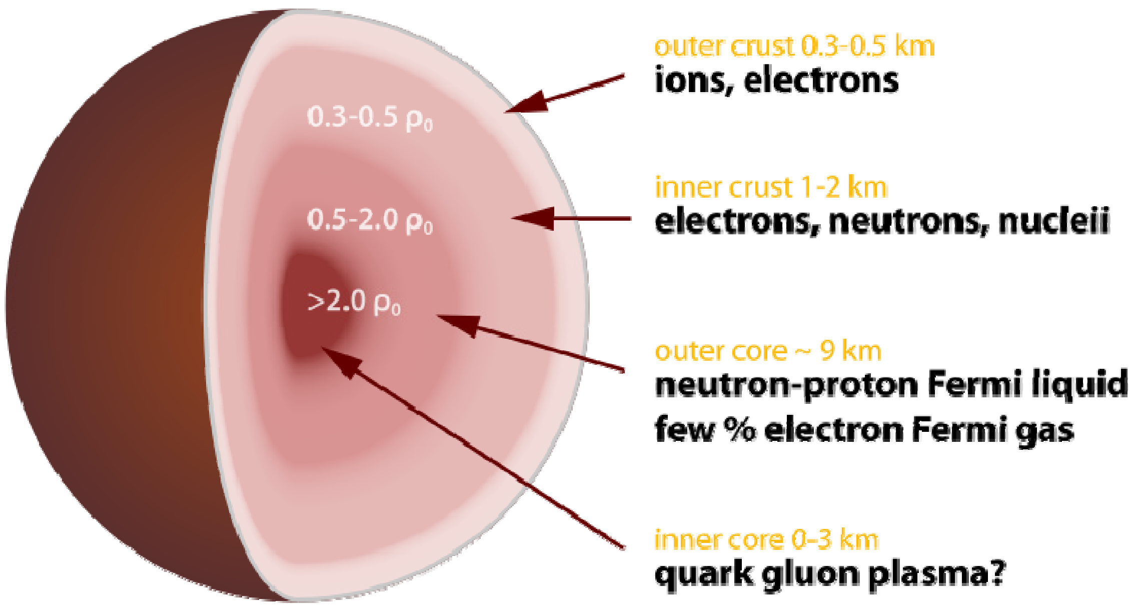


Fig. 5 Cross-section of neutron star[2]

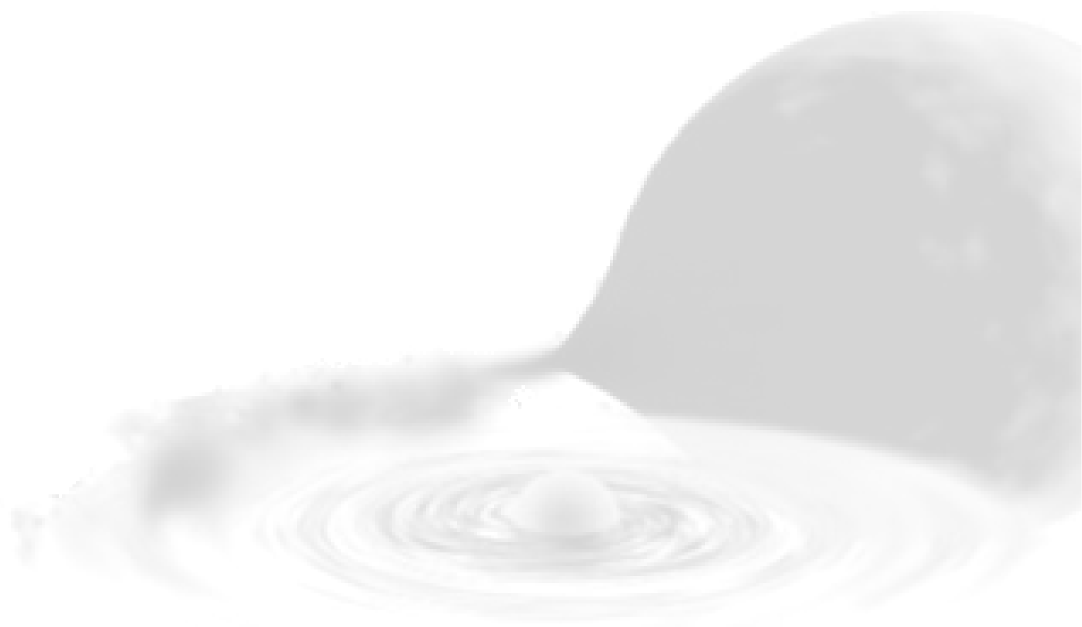
The Equation of State (EOS) of a neutron star is still not precisely known due to extremely high gravitational effects, as well as the unknown behaviour of the *exotic matter* that is believed to be composing part of the nucleus of the star (including degenerate strange matter, strange quarks in addition to up and down quarks, matter containing high-energy pions and kaons in addition to neutrons, or ultra-dense quark-degenerate matter, see Lattimer and Prakash 2004).

[2] Credit: *Robert Schulze 2010, Wikimedia Commons*. Densities are in terms of ρ_0, the saturation nuclear matter density, where nucleons begin to touch.

As a consequence several EOS have been proposed by different authors, some of them with very different predictions as to what is the behaviour of pressure (P) and density (ρ), and hence resulting in different Mass-Radius relations for the neutron star. In this regard, X-ray bursts constitute invaluable tools in determining the above mentioned properties, since (Bhattacharyya, 2006):

1. They originate from neutron star surfaces.
2. Their intensities are ~ 10 times higher than the non-burst emission intensity. This gives higher signal-to-noise ratio.
3. They show timing and spectral features, which can be used to constrain the mass, radius and spin frequency of the same neutron star.
4. They provide a unique opportunity to understand the thermonuclear flame spreading on neutron star surfaces.
5. Many bursts are observed from the same neutron star.
6. Comparatively lower magnetic fields ($\sim 10^7$-10^9 G) of the bursting neutron stars simplify the modelling.

In summary, our efforts in understanding X-ray bursts will definitely pay off because thermonuclear X-ray bursts are important for understanding neutron stars.



This Page Intentionally Left Blank

CHAPTER 2
DESIGNING PARALLEL APPLICATIONS

"Want to make your computer go really fast? Throw it out a window."

Anonymous
In L. R. Parenti, Durata Del Drammа: Life Of Drama (2005), 32.

"There are 3 rules to follow when parallelizing large codes. Unfortunately, no one knows what these rules are."

W. Somerset Maugham, Gary Montry
Quote collected by Steve Plimpton in the Massively Parallel Computing Research Laboratory at Sandia National Laboratories.

2.1. Introduction

A parallel computer is formed by a group of processors that execute specific tasks cooperatively with the goal of solving a particular computational problem. This definition is broad enough so that not only large clusters of supercomputers may be regarded as a parallel computer, but also single workstations with several processing units or a group of interconnected workstations can be regarded as well as *parallel computers*.

In the past, parallel computers and parallel programs constituted a rather obscure branch of computing, with little to none interest by most computer engineers, being only investigated by specialised researchers in the computing fields. Nowadays however, with the soaring knowledge in physics, and mathematics, together with the rapid increase in the need of computationally intensive simulations and calculations, has brought parallel computing forward, as an *enabler* or permitting *factor* of many intensive, precise simulations.

2.2. Thinking in Parallel

Consider the classical Gaussian Elimination (GE) algorithm for solving the system of equations

$$\mathbf{Ax} = \mathbf{b}\,, \tag{2.1}$$

being $\mathbf{A}$ a square matrix (NxN). The algorithm consists in adding multiples of each row to all the lower rows in order to make $\mathbf{A}$ upper triangular: $\mathbf{U}$. Afterwards, the system is solved by back substitution of the triangular system:

$$\mathbf{Ux} = \mathbf{c}\,. \tag{2.2}$$

In the following figure, it is shown the first part of the algorithm in its sequential form.

That is, after N-1 iterations matrix $\mathbf{A}$ is reduced to an upper triangular matrix U. This forward elimination process takes

$$\frac{N^3}{3}+\frac{N^2}{2}-\frac{5N}{6} \tag{2.3}$$

floating point multiplications and

$$\frac{N^3}{3}-\frac{N}{3} \tag{2.4}$$

floating point additions (Tapia *et al.* 2001). It is easy to see that with a computational cost proportional to the third power of the dimension of $\mathbf{A}$, the time it takes to solve the system grows exponentially with N. A way to reduce computing time is to distribute the forward elimination work amongst a group of P processing nodes (0, …, P-1), so that computation work is performed in parallel.

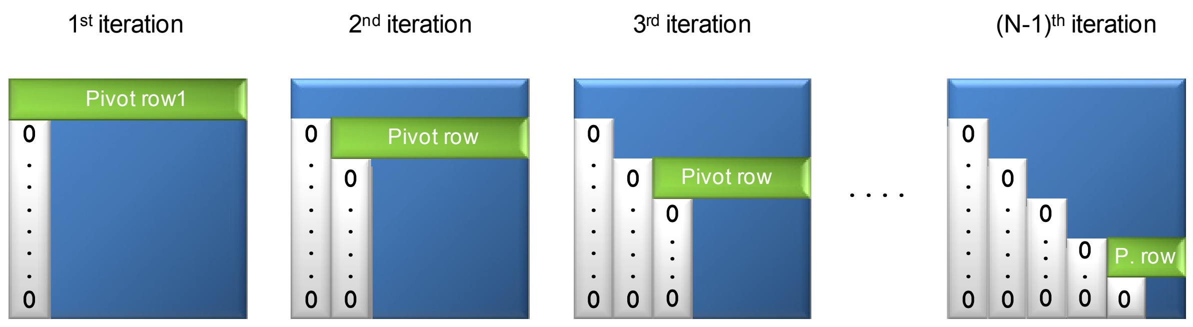


Fig. 6 Gaussian elimination algorithm representation

One such parallel Gaussian elimination algorithm is described in McGinn *et al.* (2002) and depicted in Fig. 7. In this algorithm, matrix A is centralised in process 0 (root) which centralises and coordinates the communication work with the rest of the processing nodes. Process root (P_0) sends (broadcasts) the pivot row to all other processors that collaborate in the computation. Afterwards, all rows below the pivot row are grouped in *chunks* of data that are subsequently sent to the processors. Each process receives its own chunk of data. Note that the number of rows has to be split homogeneously amongst processors so that the workload is equally distributed.

At this point, each process adds multiples of the pivot row to the rows in their chunk of data, so as to zero out all elements below the diagonal (first column). This is a purely parallel phase of the computation, as each process can proceed in parallel with no need of communications. Once each process has performed the computations, the modified chunks of data are sent back (gathered) in Process 0, which updates matrix $\mathbf{A}$ accordingly. The process repeats again, now with pivot row 2, whilst the data chunks diminish as the matrix reduction progresses.

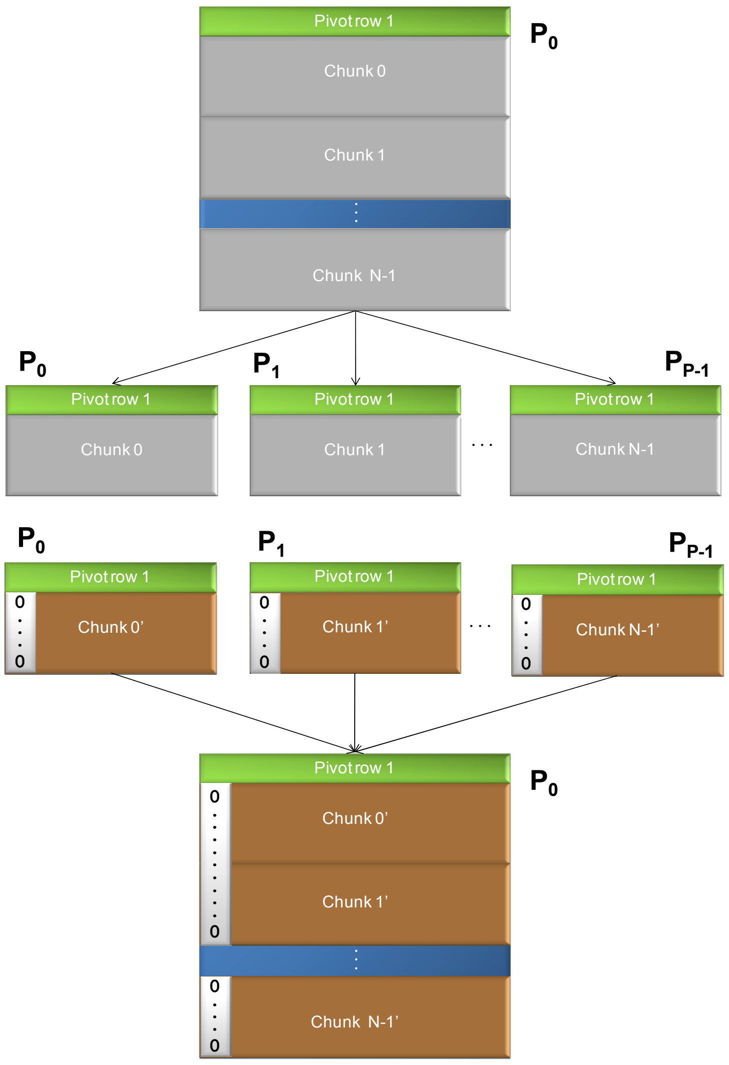


Fig. 7 Parallel Gaussian elimination algorithm

More examples of parallel algorithms can be found in Foster (1995), and Lafferty (1993). MPI Forum (2009) also includes illustrative examples using the message passing interface standard for communications (see section 2.3.4).

2.3. Designing and Building Parallel Programs

2.3.1. Parallel Computer Model

Computer scientists tend to classify machines in different ways to provide different insights into the machine's architecture. One well-known approach developed by Flynn (1972) classifies a machine by the type of instruction stream (a sequence of instructions performed by a computer) and data stream (a sequence of data that the computer performs instructions upon). Following Flynn's format, machines can be designated as either single instruction stream, multiple data stream (SIMD) or multiple instruction stream, multiple data stream (MIMD) machines (Lafferty, 1993).

In the parallelisation discussed herein, our approach is mainly SIMD, as several nodes execute (mainly) the same nucleosynthesis code, but with two different approaches:

1. With completely different, independent data at different stages of the computation (for instance when the nucleosynthesis code is coupled to a hydrodynamic code, in which case each node can compute nucleosynthesis on different disjoint layers of the neutron star's envelope).
2. Cooperatively, working with different chunks of data from the same main data set (this would be the case of the post-processing calculation, in which all processors work cooperatively on computing the nucleosynthesis within the same envelope's shell).

In addition, a second approach commonly used by researchers to classify machines is by the memory architecture of the machine. For example, shared-memory multiprocessors (SMP) imply multiple processors sharing one common memory, whereas distributed memory multiprocessors (DMP) imply that memory is localized to each processor. These terms will also be used extensively throughout this report. Although a high-performance computer may only have one processor, the emphasis of this report is on those clusters of computers that utilise multiple processors which run concurrently, thus the term parallel computing.

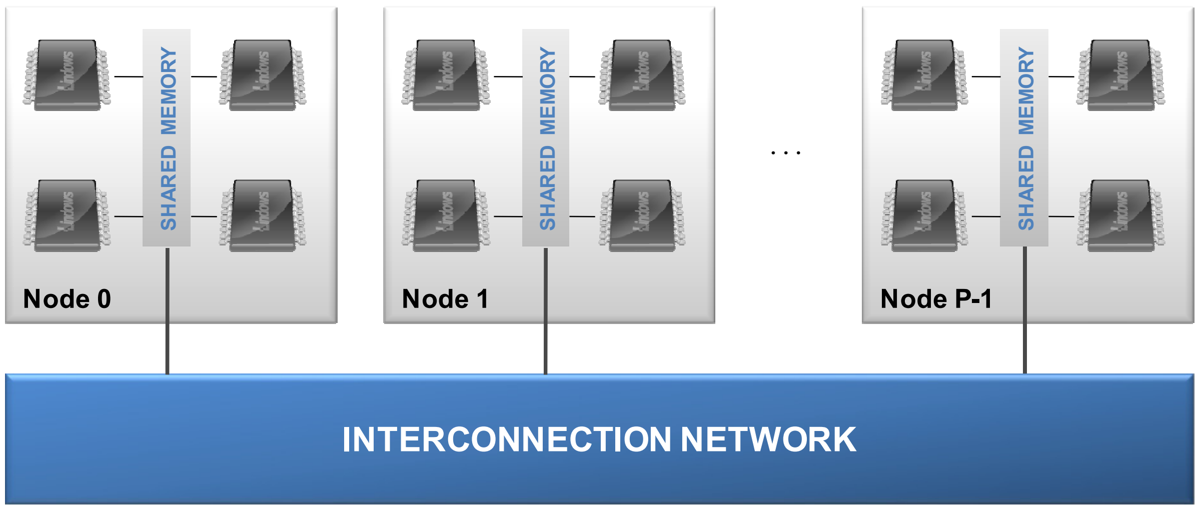


Fig. 8 Cluster architecture with shared and distributed memory multiprocessors

Shared memory multiprocessors have the clear advantage of localisation; sharing a common memory *usually* means that processors are localised on a single node or workstation (this is not *always* true, as there could exist shared memory multiprocessors with a common memory allocation *outside* the nodes, although in this case the benefits of proximity are lost). In the most usual case that shared memory also means shared location, message exchange between the processors will be significantly fastest than for their distributed memory counterparts, since intra-processors communications are much more effective than inter-processors communications (networked communications). This is due to the fact that in a distributed memory environment, communication latency significantly increases, as messages exchanged between processors are transmitted through Ethernet switches, and network interfaces. All such communication equipment introduce both physical delay (due to *material* physical properties of the interconnection network, such as distance between nodes, number of nodes, speed of transmission channels, etc.), as well as processing delay (due to particularities of the transmission protocol, such as error checking, acknowledgments, packet distribution, etc.)

Fig. 8 shows a cluster architecture showing both shared and distributed memory multiprocessors which are connected through an interconnection network. The cluster depicted consists of a group of P interconnected workstations in which each workstation consists of several processors. This is the environment where our parallelisation work has been carried out, as described in the following section.

2.3.2. Granularity Levels of Parallelism

The performance or speed-up accomplished in the parallelisation hinges on the nature of the problem being parallelised. Particularly, how data is accessed and processed gives information about how painless or painful it will be. Classifying the problem at hand into several categories (problem architectures), one can determine the likeliness of achieving good performances and whether or not it is worth the effort (Fox, 1991).

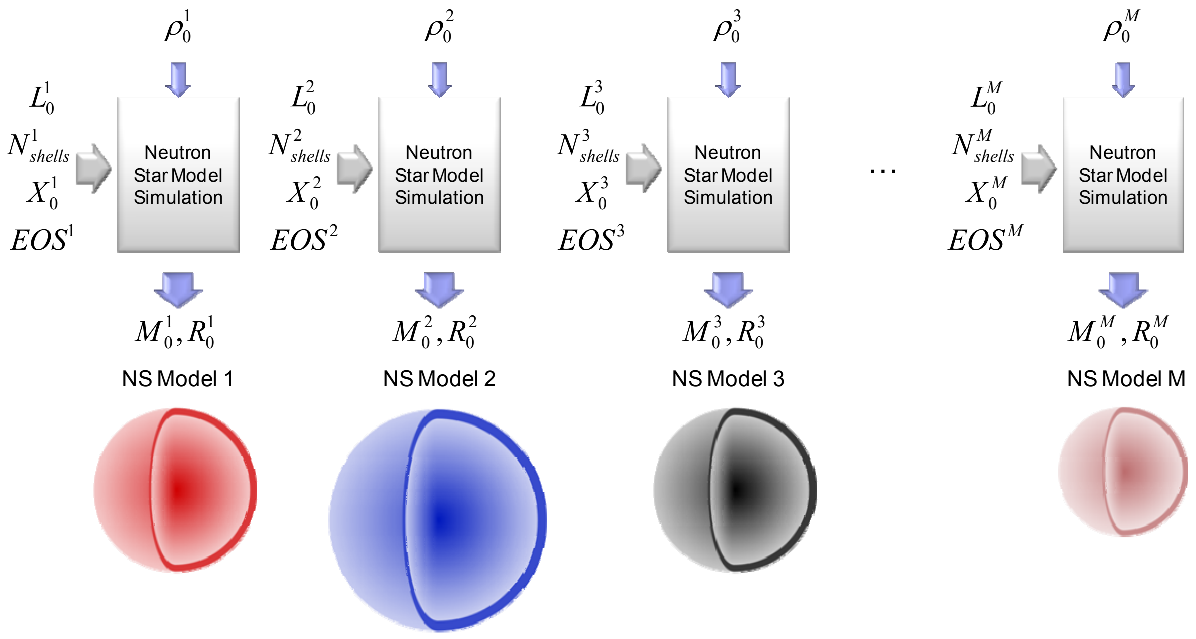


Fig. 9 An example of perfect parallelism: a set of initial neutron star models

Consider the problem of generation of several neutron star models depicted in Fig. 9. Given the central density (ρ_0), the number of shells (N_{shells}), the initial luminosity (L_0), the initial chemical composition (X_0) and the chosen equation of state of matter, electrons, ions and radiation (EOS), one can construct (by integration of the set of equations of conservation) an initial model of the neutron star in hydrostatic equilibrium, yielding, amongst other data, the mass and radius of the star (Shapiro & Teukolsky 1983). The simulation can be a sequence of serial executions, each calculating a neutron star model from different chosen parameters; or parallelism can be introduced if we process multiple input parameters at the same time, in order to generate several stellar models.

This is obviously the most desired problem to parallelise, and the most wanted by parallel programmers worldwide. This problem architecture is referred to as *perfect parallel*. Basically, the calculations on each input parameters can be executed fully independent from each other, running copies of the code on several machines, provided that each copy has the appropriate input data for their simulation. Speed-ups of the order of M (being M the number of processors or machines where the simulations are run into) are easily achievable. This is why this style of parallelism is often called "embarrassingly parallel" because it is embarrassingly easy to parallelise.

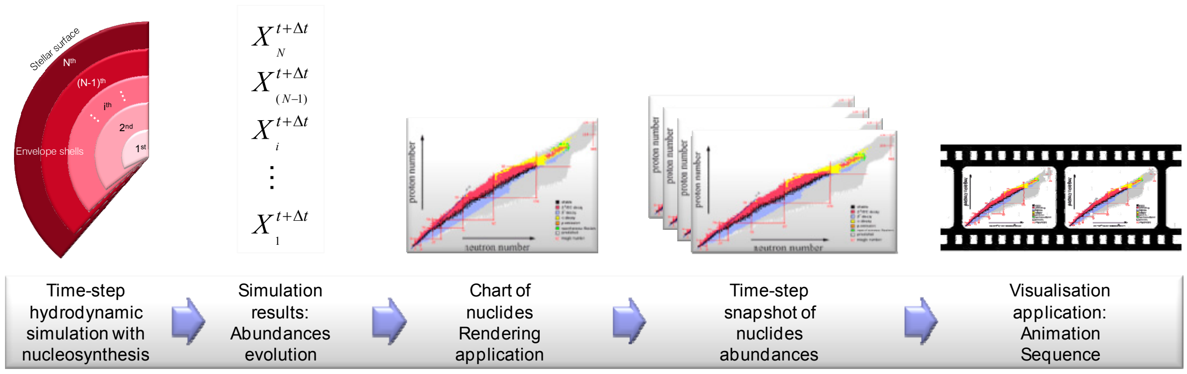

Stellar surface
Nth
(N-1)th
ith
2nd
1st
Envelope shells
$X^{t+\Delta t}_{N}$
$X^{t+\Delta t}_{(N-1)}$
$X^{t+\Delta t}_{i}$
$X^{t+\Delta t}_{1}$
proton number
neutron number
Time-step hydrodynamic simulation with nucleosynthesis
Simulation results: Abundances evolution
Chart of nuclides Rendering application
Time-step snapshot of nuclides abundances
Visualisation application: Animation Sequence

Fig. 10 An example of pipeline parallelism: a chart of nuclides animation

Now imagine a different scenario where data to be processed are not completely independent; for instance if every data set has to be processed in a series of time steps as shown in Fig. 10; a chart of nuclides evolution is animated so that the evolution of the several abundances is visualised in an movie-like sequence. In a series of time-steps a hydrodynamic simulation coupled with a nucleosynthesis code calculates the time evolution of the abundance of the isotopes of the considered reaction network. The list of nuclides and their relative abundances are then formatted for being displayed in a chart of nuclides diagram. Finally, a visualisation application gathers the different snapshots and animates the sequence for visualisation. Should this application be carried out sequentially, the nuclides abundances at each time step would serve as input to the chart of nuclides rendering program, whose output would in turn be used as input to the animation application. A quick analysis of the problem reveals that parallelism can be exploited if the several processing stages are overlapped so that the rendering of chart of nuclides is started as soon as the abundances at the first time step are available. Afterwards, as the hydrodynamic code produces the third set of abundances, the chart of nuclides rendering proceeds on the second data set, whereas the first abundance set is then animated and displayed (Fig. 11).

This problem model is referred to as *pipeline parallelism* (Pancake 1996), because data sets are in effect "piped" from one processing step on to the next. The key point is that processors can work independently on consecutive data sets as long as data sets are passed just one way through the pipe (that is, the hydrodynamic code does not require information from the chart rendering or visualisation applications). The start of the execution is initially delayed as the data set becomes gradually available to the several processing stages, so the gain in performance due to parallelism will depend on the number of processing steps that can be effectively run in parallel.

However, this type of parallelism poses evident problems. If several processing stages are not computationally equivalent, faster phases will run quicker than the slower ones, so that processors executing them will finish the execution and remain idle, waiting for more work. A possible solution is to take into account the characteristics of the several nodes involved in the simulation, hence assigning the most computationally intensive tasks to the ones with faster CPUs. At any rate, this would prove a difficult task, and its effectiveness lays strongly on the nature of the problem being parallelised. All in all, pipeline parallelism is not as simple or efficient as perfect parallelism described above.

There are many more applications where results cannot be obtained in a one-way processing flow between stages. Consider, for instance the fully coupled hydrodynamic code example depicted in Fig. 17. The evolution of the physical values of each of the envelope's shells is affected by the evolution of the other, mainly neighbouring zones of the envelope. For instance, temperature gradients may show convection setting in, with the subsequent disturbance propagation towards neighbouring shells. If the simulation were to be executed sequentially, the calculations would need to be performed across all the data on all shells to find a specific envelope's state, and then a new iteration would begin. Parallelism may be introduced with multiple nodes or processors participating at each time step, where each processor would take mainly the processing work of some of the shells. Every single iteration has to be completed across all data before the next time-step begins.

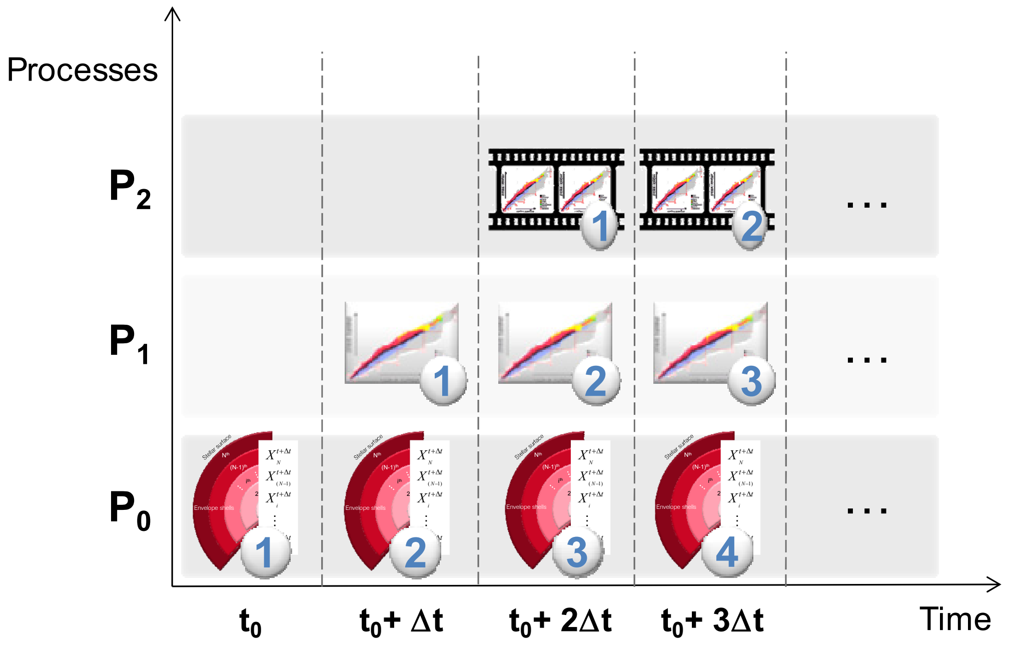


Fig. 11 An example of how a pipeline parallelism is executed on three processors

This problem architecture represents the so called *fully synchronous parallelism*, indicating that (at least in principle) each computation is performed synchronously (or simultaneously) to all data. The main point here is that all future calculations of decisions hinge on the results of the earlier, preceding data calculations. There are usually not enough processors to execute a computation to all data of all shells at the same time, therefore each node actually cycles through a subset (i.e. a number of contiguous shells). If this group of shells, assigned to each processor, is not homogeneous, the workload may vary across different nodes. This is often the case where a disturbance in a specific shell starts to propagate to upper layers, modifying the physical and compositional variables of these upper shells, whereas lower layers may rest unaffected. A clear example of this is the buoyancy forces acting on a bubble of hotter matter, moving the material upwards to an area of lower pressure, and exchanging in the process heat and mass as it proceeds to upper layers (Kippenhahn *et al.* 1996). As a consequence, if each process executes computations on a group of contiguous shells, only nodes acting on the area containing the shells affected by the disturbance would perform intensive work at this point. In the meantime, synchronicity requires that all the other processors cannot continue with the next set of computations, so they must wait for the busier ones to catch up. Consequently, fully synchronous parallelism stresses the programmer skills more than pipeline parallelism in the bid of attaining good performance.

Consider now a post-processing nucleosynthesis code whose computing flow is shown in Fig. 17. From a complete temperature-density versus time profile, previously calculated in a hydrodynamic code coupled with a reduced nucleosynthesis network, the application calculates the evolution of the abundances of the selected nuclides. Temperature and density are interpolated at the current simulation time; this allows determining the value of the reaction rates driving the nucleosynthesis at the current temperature and density. After this, the linearised system of equations of the network abundances derivatives is constructed and solved for proper precision. At this point of the simulation convergence and stability checks are performed, and, if failed, the chosen Δt is reduced so that the system of equations has to be build up again from scratch. Only if the convergence criteria determine that the simulation time-step leads to convergence, the algorithm moves forward to calculate the nuclear energy generated.

This scenario is a clear example of a *loosely synchronous parallelism*. None of the processing stages can be executed in parallel with the others, because all of them need the outputs of the previous steps to do their computations. For instance, it is not possible to build and solve the system of equations that arises from the linearisation of the set of differential equations describing the temporal evolution of the network abundances, if the reaction rates have not been previously provided. Likewise, nuclear energy generation can only be computed once the abundance variations have been computed and the convergence and stability requirements met. As a consequence, this problem type can only be parallelised if all processors contribute to all computing stages of the simulation, exchanging information whenever needed. Over and above, when each simulation step ends, processors that have finished their computation work must wait until all the other nodes have completed their work too. This is due to the fact that they must share their intermediate results before going on to the next time step. Loosely synchronous parallelism, suffers from the downsides of both pipeline and fully synchronous parallelism, which makes it the least amenable problem type to being parallelised. With loosely synchronous parallelism it is hard to equally distribute evenly computation work between nodes. In particular (regarding the post-processing nucleosynthesis code parallelised in this Thesis), it will be described in section 3.3.3, the devised strategy to build the matrix network in a distributed manner, so that each process constructs only a portion of the complete system of equations (the chunk it will later need to solve their part of the system of equations). It will be shown later on, that, given the sparse structure of the network matrix $\mathbf{A}$, it is highly difficult to evenly allocate the work to several nodes for the resolution of the system.

2.3.3. When is Parallelisation Effective?

Parallelism is not achieved without a cost. There is a steep learning curve to parallel programming, as well as requiring considerable effort from the programmer, who must think on the problem in completely new ways and may wind up rewriting almost all of the sequential (single-node) code. In addition, parallel execution and development environments are inherently unstable and, at times, lacking of deterministic behaviour. Completely different techniques and strategies are used in the parallel world from those used to optimise and tune the performance of single-node, sequential applications. Debugging a parallel application is considerably harder than their sequential counterparts (Pancake *et al.* 1994, McGraw *et al.* 1998).

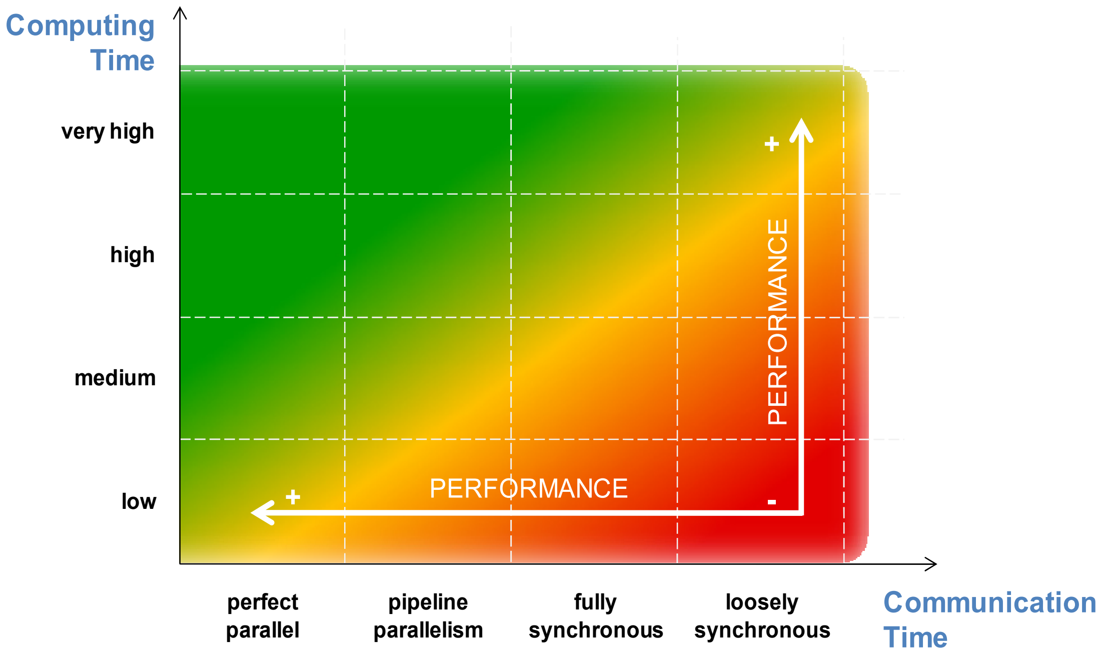


Fig. 12 Performance of the parallel problem architectures

To take the most out of parallelism it is therefore needed to analyse first the nature of the problem at hand, and identify the problem architecture it may fit into (see section 2.3.2), in order to clearly anticipate whether parallelism is worth it. According to Pancake (1996), this can be determined applying the following four rules of thumb:

1. If the application can be classified as a perfect parallel problem, the parallelisation work will be acceptably straightforward and good performance is very likely to be achieved.
2. If the application fits the model of a pipeline parallelism, more work has to be put into the parallelisation tasks, taking into account that the key to attain good speed-ups is to balance the computational intensity.
3. If the problem at hand is identified as fully synchronous, a significant amount of work is required and it might not eventually pay off. A decision has to be made according to how evenly (equally distributed) the computational intensity will be.
4. The worst possible scenario is that of a loosely synchronous application, which it is the most difficult problem to parallelise, by far. It is not worthwhile unless the ratio between computation work and communication time is maximised (the points where the nodes interact must be very infrequent).

All of the aforesaid guidelines to determine whether parallelisation is worthwhile or not, can be summarised in a simple principle: the time spent on communication between processors has to be kept to a small fraction of all computing time, that is, the main goal of any parallel programmer will be to maximise the ratio:

$$R = \frac{Computation\ time}{Communication\ time}\ . \tag{2.5}$$

The best approach to increase performance is thus the programmer's skills both to keep communication to a minimum, regarding the interaction points and the data being transferred, and to keep the processors busy in the distributed computations. For the first type of applications discussed above (perfect parallel applications) this is relatively trivial, but fully synchronous and loosely synchronous applications will only achieve acceptable speed-ups if there are little interaction points between processors and/or long periods of time in which the processors are allowed to exchange their data.

This is shown in Fig. 12, where it is depicted a qualitative measure of the performance achievable for the several problem architectures and problem sizes. Green zones represent substantial speed-ups, whereas yellow to orange zones represent moderate to small increases in performance. It is illustrative to stress that applications falling into the red areas will most certainly take more time to complete in their parallel version than in their sequential counterparts, and therefore the viability of parallelisation has to be carefully evaluated in advance. Note that the larger the problem size is, the longer it will be the time the processors spend on computations. Similarly, the four types of problem architectures have an increasing need of communication, and hence communication times will be the shortest for perfect parallelism and the longest for loosely coupled parallelism. Fig. 12 reveals that the only way to achieve respectable performances on a fully or loosely synchronous application, is to increase the ratio (2.3), increasing the problem size (e.g. increase the problem resolution or complexity), so that processors spend more time on computation.

The strategy should hence be quite clear: parallel applications must be designed to execute almost independent processors. The less frequently the processors establish communication (either by specific operations, blocking communications or explicit messages), the better the application's scalability and performance will be. That is, in order to achieve high parallel performance and scalability, one must strive for embarrassingly parallel algorithms, either by the careful design of data structures and/or algorithms, the utilisation of parallel applications and environments already existing, or finally by turning the problem into an algorithm for which a perfect parallel solution exists (McKenney 2010)

Fully and loosely synchronous applications are more suitable to be executed on Shared Memory Multiprocessors (SMPs, see section 2.3.1), because the high cost of network communications between distributed memory multiprocessor may take too high a toll on communication time for fully and loosely synchronous applications to be effective. In extreme cases, especially if the problem is medium to small sized, communication time of the parallel application may overtake the time it takes a serial program to be executed in a single machine. It will be shown later on, that this is the case of a post-processing nucleosynthesis parallelisation with a medium sized network of nuclides (~600 nuclides).

Finally, the gains in performance cannot be evaluated independently from the amount of resources (CPUs) needed to achieve them. If a parallel version of a program is able to run a simulation four times faster than the serial version, the speed-up is excellent if, say, five to six nodes (CPUs) are used in the computation, but we would be wasting resources if we needed twenty nodes to achieve this very same increase in performance. At the end of the day, it all boils down to strike a balance between performance, application type, and needed resources.

2.3.4. Parallel Programming Models

There are several parallel programming models that can be used to describe how the parallel program executes, and to model aspects such as modularity, scalability, and performance of the parallel program (Foster 1995). There is no programming model that suits all types of parallel applications, but rather we must choose the programming model together with its implementation that better fits our problem at hand as well as the parallel resources available to us.

In the *Tasks and Channels* programming model, a computation consists of a set of tasks connected by communication channels. A task is used to model encapsulation of a program that executes with local memory, and defines an interface to other tasks for communication. A channel is just a message queue used to place messages to and from other tasks. The *Message Passing* model is a minor variation of the tasks and channels model, where each node executes one or more tasks that communicate with the other processors by means of message passing (MPI Forum 2009, Gropp *et al.* 1995). All processors execute the same code, but with different data, therefore message passing is a *single program multiple data* (SPMD) programming model. This is one of the most widely used parallel models, and has been the chosen paradigm for the parallelisation of the post-processing code in this Work. *Data Parallelism* is another model commonly used in parallel applications (Koelbel *et al.* 1994, Zima 1991). Data parallelism exploits the concurrency that may derive from the application of the same operation to multiple elements of a data structure. Its application is therefore limited by the nature of the problem being parallelised, and may be limited to single operations on arrays or data structures that may be encapsulated in a bigger parallel framework. Finally, the *Shared Memory* programming model establish that parallel jobs access a common address space, which the several tasks use to read and write data in an asynchronous way (Gottlieb 1983, Snyder 1986). Consequently, concurrency control mechanisms like locks and semaphores must be put in place to control coherent access to shared memory locations. One clear advantage of this model is that tasks do not need to communicate in a message exchanging fashion; however, the management of locality becomes clearly harder. Also, its application is limited to shared memory multiprocessors (section 2.3.1), where it is easier to have multiple processors sharing a common memory space.

In the parallelisation performed in this Thesis, it has been used the MPICH2 implementation of the Message Passing Interface (MPI) standard (MPI Forum 2009, MPICH2 2011, and Appendix A). In the MPI programming model, the execution takes place across one or more processors that communicate by making calls to MPICH2 library routines, in order to send and receive messages from other concurrent processors. In this implementation (as in most of the MPI implementations), the set of processors is fixed at start-up, being one process created per processor. Processors can use point-to-point communications and send a message to a specific process; this can be used to implement communications in a local or unstructured way. Other commonly used communication mechanisms are the so called *collective* communications, which can be used to collectively send and receive information (broadcast, summations, gathering of intermediate results, etc). MPI supports both synchronous and asynchronous communications; this makes it possible so manage effectively the time spent by processors waiting for communications to complete.

Chapter 3 provides the details of the message passing parallelisation strategy designed in this Thesis. Also, Appendix A, presents the software tools needed to build, compile, and debug the parallel application.

2.4. Performance Limits

Computer scientists study parallel programming as a research subject in itself, but that is not the generalised interest of most scientists and engineers. It is not parallelism what is looked for, but the gain in performance that it brings about. However, if using multiple CPUs to run a task does not accomplish results far sooner than a single CPU, we will all agree that computing resources are being used inefficiently. Over and above, an engineer has had to spent time and effort parallelising the application, so that human resources would have been wasted too.

To avoid that problem, it is necessary to assess the application's potential for parallelisation, and the kind of performance that is achievable for the application at hand. The speed-up is usually measured as the ratio between the time it takes the application to run in 1 node (serial execution), and the time it takes to execute on several nodes (parallel execution):

$$\text{Speed-up} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{2.6}$$

This assumes that there exists a serial program that already runs the computation in a sequential, one node version; this serial version of the code is referred to as the *baseline* for parallel performance measures. Strict supporters of parallel programming claim that a parallel application must be constructed from scratch, but this represents an unrealistic situation for the majority of the users (several surveys on the development of parallel applications indicate that about 60% of programmers modify or develop parallel programs from existing codes, whereas only about 30%, mainly computer scientists or applied physicist or mathematicians, start the parallel program from scratch - Pancake *et al.* 1994).

Not all the serial program's contents can be parallelised. Usually, initialisation and output phases must be executed sequentially (where initialisation files might be read in, or when output files have to be written to disk). Consequently, we may represent serial execution time by:

$$T_{\text{serial}} = T_{\text{initialisation}} + T_{\text{pp}} + T_{\text{output}} \tag{2.7}$$

where T_{pp} is the time spent in the part of the code that is potentially parallelisable. In the post-processing nucleosynthesis code object of this Thesis, $T_{\text{initialisation}}$ and T_{output} are negligible with respect to T_{pp}, and therefore we can safely approximate $T_{\text{serial}} \approx T_{\text{pp}}$.

Consider the parallel example depicted in Fig. 7; it is evident that the time spent in the parallel version of the code would be:

$$T_{\text{parallel}} = T'_{\text{initialisation}} + T_{\text{cp}} + T_{\text{comm}} + T'_{\text{output}} \tag{2.8}$$

where T_{cp} is the time spend by each of the processors in the execution of their portion of the problem, and T_{comm} is the time devoted to communications and message passing amongst nodes. Also, in this situation initialisation and output phases of the parallel version ($T'_{\text{initialisation}}$ and T'_{output}) are totally negligible with respect to $T_{\text{cp}} + T_{\text{comm}}$ so they can be safely omitted from the theoretical approximation of the achievable speed-up; $T_{parallel} \approx T_{cp} + T_{comm}$. Furthermore, the individual computing time T_{cp} can be approximated by:

$$T_{cp} = \frac{T_{serial}}{P} + T_{overhead} \tag{2.9}$$

where P is the number of nodes participating in the parallel computation, and $T_{overhead}$ is the overhead time spent managing parallelisation (message construction and storage, sender-receiver synchronisation, initialisation of parallel subroutines, imperfect concurrency, etc). This might be a rough estimation, as we have assumed that the parallel portion of the code is perfectly parallelisable so that it can be cleanly split up (with no overlapping tasks) into P processors. This might not be the general case, but it is a close approximation to the real execution time, and it will serve the purpose of providing an estimation of the accomplished (or attainable) maximum speed-up, which can finally be approximated by:

$$\text{Speed-up} \approx \frac{T_{serial}}{T_{cp} + T_{comm}} = \frac{T_{serial}}{\frac{T_{serial}}{P} + T_{overhead} + T_{comm}} = \frac{1}{\frac{1}{P} + \frac{T_{overhead}}{T_{serial}} + \frac{T_{comm}}{T_{serial}}}\,. \tag{2.10}$$

Since the overhead time will be usually much smaller compared to the serial execution time ($T_{overhead} << T_{serial}$), the maximum attainable speed-up will be driven by the ratio between T_{comm} and T_{serial}, so that the higher the ratio, the smaller the speed-up accomplished with the parallelisation. Note that this result comes into agreement with the conclusions of section 2.3.3, where we found that for higher speed-ups, the ratio between processing time and communication time had to be maximised.

It is worth mentioning that both T_{comm} and $T_{overhead}$ vary with the number of nodes, that is:

$$T_{comm} = T_{comm}(P), \quad T_{overhead} = T_{overhead}(P)\,. \tag{2.11}$$

The amount of variation with P highly depends on the type of communication (e.g. all to all, broadcast, point to point sends and receives, gather, all gather, etc. – see *MPI Forum: 2009*), but at any rate they are both monotonically increasing functions of P, with a much more pronounced variation of T_{comm} than $T_{overhead}$ for increasing values of P (Thakur *et al*, 2002).

The effect of the costs of communications with respect to the total (serial) execution time, is analysed in Fig. 13 below, for increasing values of T_{comm} given a fixed T_{serial} and increasing number of parallel nodes involved in the execution. The blue straight line with a unity slope represents the ideal speed-up: $\text{Speed-up} = P$, which is obtained by setting to zero both communication and parallel overhead times. That is, in an ideal situation, putting P processors into a parallel task would yield results P times faster compared with the serial execution. This will obviously not be the case that we will find in a real application. As communication and overhead times become not-negligible, the curve looses slope and bends rightwards. Green and yellow lines represent a more real situation with low and medium communication costs, respectively. The green line could easily be the speed-up accomplished in a perfect parallel application, whereas the yellow line may represent the performance of a pipeline or fully synchronous parallelism (see section 2.3.2). Too main conclusions can be drawn from this two cases; firstly, as we increase the number of nodes, the efficiency of the parallelisation decreases (the curve's slope diminishes), so that every new processor added into the computation, accounts for smaller and smaller percentage of the total speed-up achieved. Secondly, it is important to note that in a real application, the maximum achievable speed-up is finite, that is, even if we could use infinite processors in the parallel application, the final speed-up tends asymptotically to finite, fixed value.

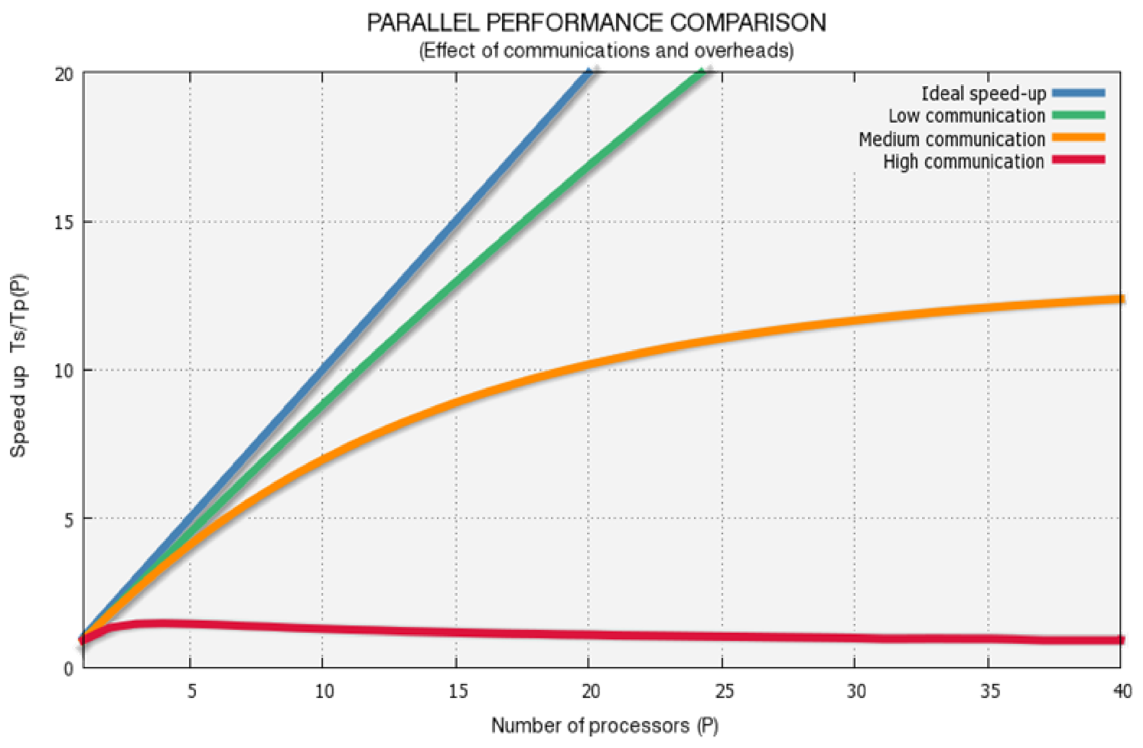


Fig. 13 Parallel performance comparison for different values of communication costs

The worst case is represented by the red line in Fig. 13, which represents a loosely synchronous application with a high communication-to-processing time ratio. Note that for a small number of processors, the benefits of the parallelisation may outrun the burden of the high communication costs, but as we increase the number of processors (and hence communication times), the communication and parallel management tasks rapidly outgrows the time saved up in the parallel execution, yielding speed-ups that may actually fall below unity, that is, the parallel application may take longer to execute than its sequential counterpart.

2.5. Execution Environment: Hyperion Cluster

The development environment where this Thesis has been developed consists of a 42-node Cluster (Hyperion) that the Astronomy and Astrophysics Group (GAA) has at the EUETIB (UPC). All the nodes have local memory so that the architecture is that of a distributed memory multiprocessor. Each of the 20 machines of the cluster is in itself a multiprocessor workstation with 2 to 4 cores that correspond to the shared memory multiprocessor environment described above. In total there is one 4-processor machine and 19 machines consisting of 2 processors each.

Note that in the remaining of this document, we will regard each single core or processor as an independent processing unit, so that the terms node, processor, and core all refer to a single microprocessor unit. Therefore, in our cluster environment there is a maximum of 42 nodes or processing units (4x1 plus 19x2).

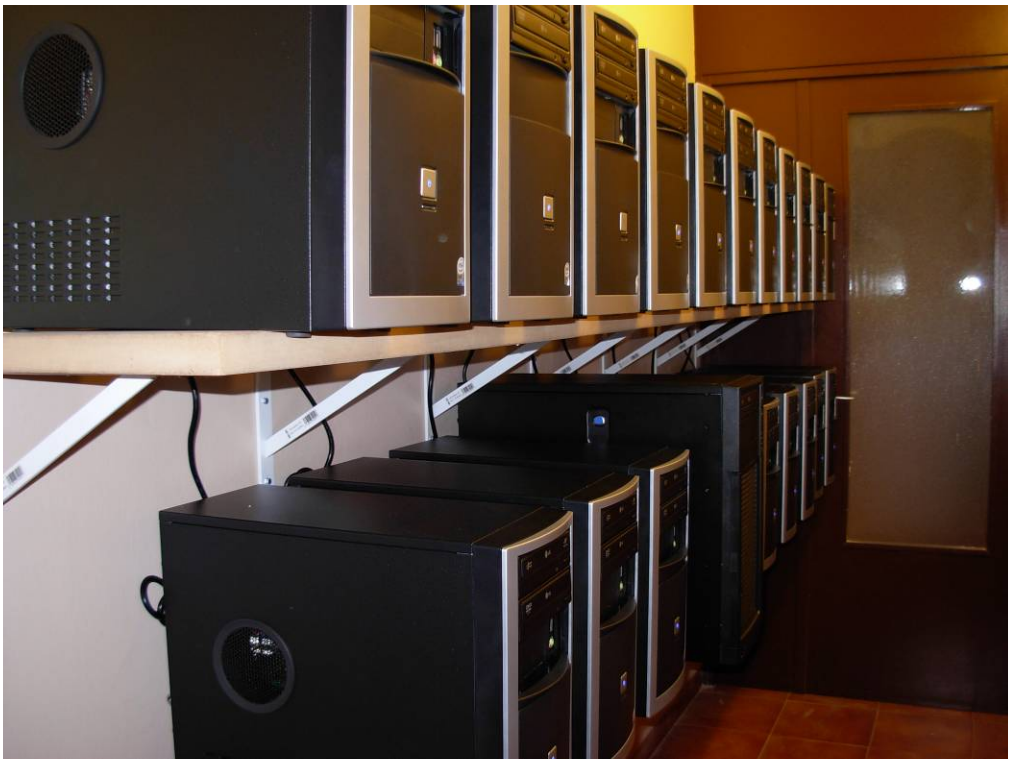


Fig. 14 Hyperion cluster

Due to the high efficiency requirements, clusters are managed by a dedicated operating system that (amongst other things) handles all communication issues between nodes or workstations. The Hyperion cluster has the Rocks Cluster Distribution operating system (http://www.rocksclusters.org/), which is a Linux distribution intended for high-performance computing clusters. *Rocks* has become a widely-used cluster operating system, for academic, government, and commercial organizations.

2.6. Summary

Parallelism overcomes some of the constraints imposed by single-CPU computers, offering faster solutions, running simulations at finer resolution or modelling physical phenomena more realistically. However, parallelism does not come without a cost. Parallel programming involves a steep learning curve. It is also effort-intensive; the programmer must think about the application in new ways and may end up rewriting virtually all of the serial (single-CPU) code. Parallel performance will depend strongly on the type of application:

1. Perfect parallel: the parallelisation will be acceptably straightforward and very good performance is likely to be achieved.
2. Pipeline parallelism: more work has to be put into the parallelisation tasks. The key to attain good speed-ups is to balance the computational intensity.
3. Fully synchronous: a significant amount of work is required and it might not eventually pay off. It all depends on how evenly the computational intensity can be distributed.
4. Loosely synchronous: it is the most difficult problem to parallelise. It is not worthwhile unless the ratio between computation work and communication time is large.

Not all applications can be parallelised. If this is not taken into account, it is perfectly possible to work months on parallelizing an application, only to find that it yields incorrect results or that it runs slower now than before. Parallel applications must be designed to execute almost independent processors. The less frequently the processors establish communication, the better the application's scalability and performance will be.

CHAPTER 3

POST-PROCESSING PARALLELISATION

"Nobody wants parallelism... what we want is performance."

Ken Neves
Boeing

"Parallel machines are hard to program and we should make them even harder - to keep the riff-raff off them."

Gary Montry
Quote collected by Steve Plimpton in the Massively Parallel Computing Research Laboratory at Sandia National Laboratories.

3.1. Introduction

In this Master Thesis, a post-processing nucleosynthesis code, with a network containing 606 nuclides (H to ^{113}Xe) and more than 3500 nuclear processes, has been parallelised using the MPICH2 implementation of the Message Passing Interface (MPI) specification for the design of parallel applications using clusters of distributed workstations. This code requires (in its sequential, single-node version) about 9.1 CPU-months of calculating power to perform 50.000 post-processing calculations of X-ray burst nucleosynthesis.

One of the purposes of the parallelisation is to benefit from the 42-node Hyperion Cluster available at the EUETIB-UPC, and see whether speed-ups are achievable, in which case it would considerably provide for better and more accurate simulations (e.g. with more isotopes and reactions, or with a significant increase in the resolution, in terms of more layers of the neutron star envelope's model). The main goal is to improve the performance of the code in terms of speed, and also taking advantage, if desired, of parallel supercomputing facilities like the *Mare Nostrum* at the Supercomputing Centre in Barcelona (BSC).

Unfortunately, parallelisation does not come without a cost, and achieving speed-ups has certainly proven a difficult task. As shown in section 2.3.2, the time dependent iterations of the post-processing code, places this problem in the worst possible categories for parallelisation; that of a loosely synchronous application, where all processors have to participate throughout the iteration, exchanging intermediate results when needed. Also, due to the nature of the problem, the resulting abundances have to be broadcasted to all processors at the end of the iteration, so that they are available to every node at the next time-step, for the distributed construction of the system of equations that arises from the linearisation of the set of differential equations describing the time evolution of the abundances. This is a serious bottleneck that provokes that the simulation cannot proceed until all processors have received the results. It is easy to see that the limit of applicability will be put by the communication time; if the time spent in communications in the parallel execution is not *much lower* than the time it takes the problem to be solved in its serial version on a single machine, parallelisation does not pay off at all, and the parallelised version might even end up taking frustratingly longer to execute.

3.2. Post-Processing Nucleosynthesis Code

3.2.1. Application Description

Due to the enormous number of reactions involved in a detailed nucleosynthesis simulation for XRBs, an approach based on a pure hydrodynamical study is often computationally prohibitive. For that reason, it is quite often that it has to be adopted a post-processing approach, relying on a set of temperature and density versus time profiles (see Fig. 15). These T-ρ profiles are usually extracted from the literature or scaled to cover the wide parameter space.

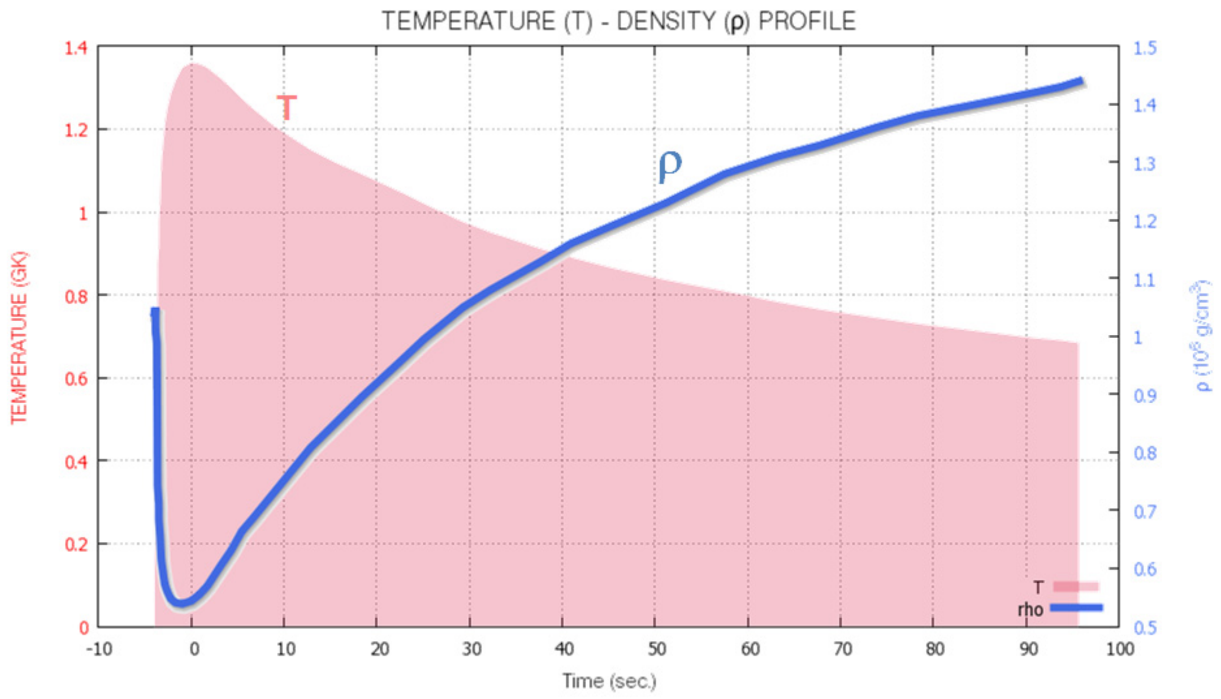


Fig. 15 Temperature vs. time profiles used in the post-processing Work of Koike *et al.* 2004

A typical post-processing nucleosynthesis code computing flow is shown in Fig. 17. From a complete temperature-density profile, previously calculated in a hydrodynamic code coupled with a reduced nucleosynthesis network, the application evolves through the simulation calculating the evolution of the abundances of the selected nuclides. Temperature and density are interpolated at the current simulation time; this allows determining the value of the reaction rates driving the nucleosynthesis at the current temperature and density. After this, the linearised system of equations of the network abundances derivatives is constructed and solved for proper precision. At this point of the simulation convergence and stability checks are performed, and, if failed, the chosen Δt is reduced so that the system of equations has to be build up again from scratch. Only if the convergence criteria determine that the simulation time-step leads to convergence, the algorithm moves forward to calculate the nuclear energy generated.

In this Thesis, it has been used a fully updated network, consisting of 606 isotopes, from ^{1}H to ^{113}Xe, and linked through a network of 3551 reactions (see Fig. 16). Elements, ranging from ^{1}H to ^{113}Xe have been marked as green squares. The location of the proton-drip line (left-hand side of the diagram), the neutron drip line (right-hand side), and the set of stable isotopes (dark grey squares) are based on Audi *et al.* (2003).

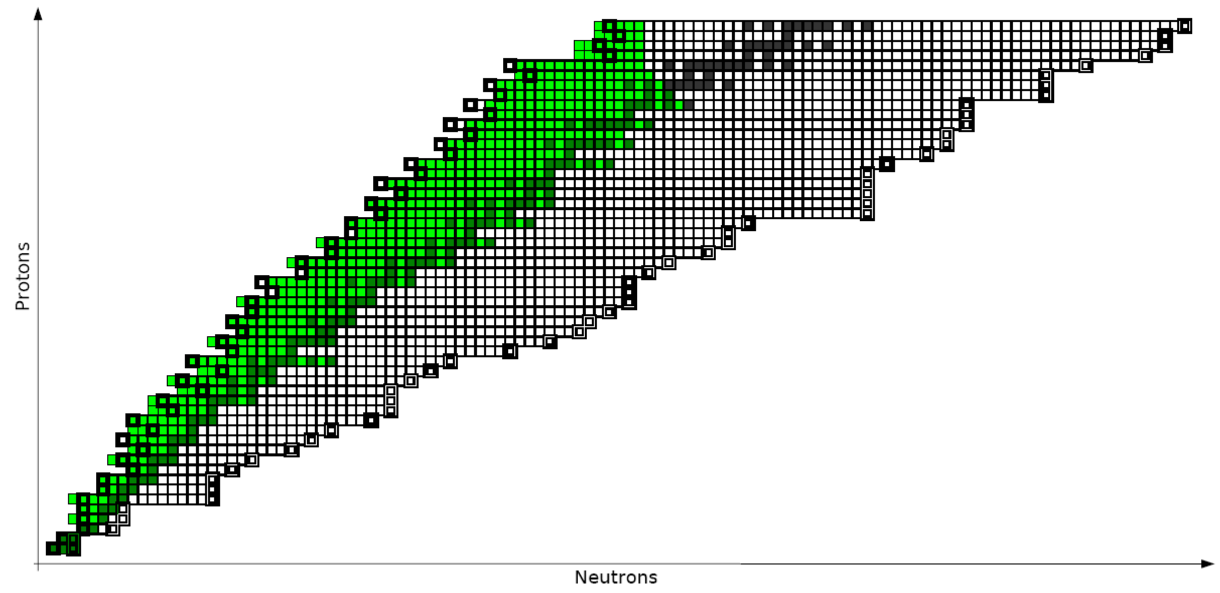


Fig. 16 Network of isotopes used in this Work for post-processing calculations[3]

By means of post-processing techniques, the time evolution of the chemical abundances is computed for a specific temperature-density versus time profile, and the set of modified reaction rates. Limitations do exist for these post-processing techniques however, since a self-consistent analysis requires putting in place a hydrodynamic code capable of self-adjusting both the temperature and the density of the stellar envelope.

Consequently, post-processing calculations are not well suited to derive absolute abundances (or to provide any insight into light curve variations and energetics) since they rely only on temperature and density versus time profiles evaluated at a given location of the star (usually, the innermost shells of the envelope). Indeed, it is likely that the evolution at other depths will be characterized by a different set of physical conditions. Furthermore, adjacent shells will eventually mix when convection sets in, altering the chemical abundance pattern in those layers. Be that as it may, this approach is reliable enough to identify the key processes governing the main nuclear activity at the specific temperature and density regimes that characterize such bursting episodes.

[3] Source: F. Moreno (2009), PhD Thesis, UPC

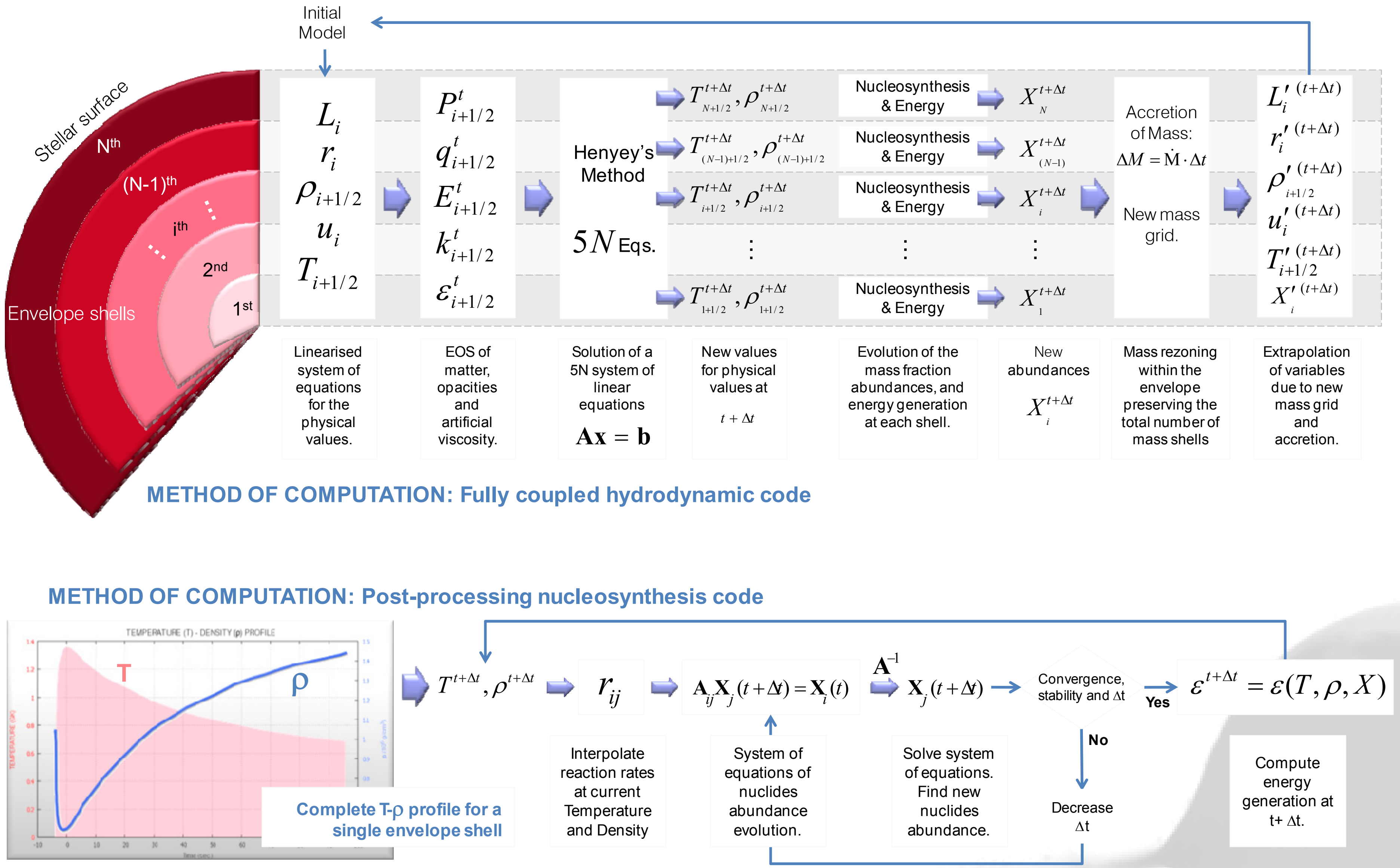

Initial Model
Stellar surface
Nth
(N-1)th
ith
2nd
1st
Envelope shells
L_i, r_i, $\rho_{i+1/2}$, u_i, $T_{i+1/2}$
$P^t_{i+1/2}$, $q^t_{i+1/2}$, $E^t_{i+1/2}$, $k^t_{i+1/2}$, $\varepsilon^t_{i+1/2}$
Henyey's Method
$5N$ Eqs.
$T^{t+\Delta t}_{N+1/2}, \rho^{t+\Delta t}_{N+1/2}$
$T^{t+\Delta t}_{(N-1)+1/2}, \rho^{t+\Delta t}_{(N-1)+1/2}$
$T^{t+\Delta t}_{i+1/2}, \rho^{t+\Delta t}_{i+1/2}$
$T^{t+\Delta t}_{1+1/2}, \rho^{t+\Delta t}_{1+1/2}$
Nucleosynthesis & Energy
$X^{t+\Delta t}_N$, $X^{t+\Delta t}_{(N-1)}$, $X^{t+\Delta t}_i$, $X^{t+\Delta t}_1$
Accretion of Mass: $\Delta M = \dot{M} \cdot \Delta t$
New mass grid.
$L_i'^{(t+\Delta t)}$, $r_i'^{(t+\Delta t)}$, $\rho'^{(t+\Delta t)}_{i+1/2}$, $u_i'^{(t+\Delta t)}$, $T'^{(t+\Delta t)}_{i+1/2}$, $X'^{(t+\Delta t)}_i$
Linearised system of equations for the physical values.
EOS of matter, opacities and artificial viscosity.
Solution of a 5N system of linear equations $\mathbf{Ax} = \mathbf{b}$
New values for physical values at $t + \Delta t$
Evolution of the mass fraction abundances, and energy generation at each shell.
New abundances $X^{t+\Delta t}_i$
Mass rezoning within the envelope preserving the total number of mass shells
Extrapolation of variables due to new mass grid and accretion.
METHOD OF COMPUTATION: Fully coupled hydrodynamic code
METHOD OF COMPUTATION: Post-processing nucleosynthesis code
TEMPERATURE (T) - DENSITY (ρ) PROFILE
T
ρ
Complete T-ρ profile for a single envelope shell
$T^{t+\Delta t}, \rho^{t+\Delta t}$
r_{ij}
$\mathbf{A}_{ij}\mathbf{X}_j(t+\Delta t) = \mathbf{X}_i(t)$
$\mathbf{A}^{-1}$
$\mathbf{X}_j(t+\Delta t)$
Convergence, stability and Δt
Yes
No
$\varepsilon^{t+\Delta t} = \varepsilon(T, \rho, X)$
Interpolate reaction rates at current Temperature and Density
System of equations of nuclides abundance evolution.
Solve system of equations. Find new nuclides abundance.
Decrease Δt
Compute energy generation at t+ Δt.

Fig. 17 Hydrodynamic code versus post-processing nucleosynthesis computation

3.2.2. Time Evolution of the Nuclear Abundances

A typical nuclear reaction may be described by two particles, P_i and P_j, which mutually interact, producing a pair of particles, P_k and P_l, in the form (Wagoner 1969):

$$^{Z_i}_{A_i}P_i + ^{Zj}_{A_j}P_j \rightarrow ^{Z_k}_{A_k}P_k + ^{Z_l}_{A_l}P_l$$

where Z_i and A_i are the atomic and mass numbers of particle i, respectively. Nuclear reactions are governed by the standard laws of conservation of energy, linear and angular momentum, mass number, and charge. The time evolution of species i is then computed from a detailed balance between reactions that create and destroy such isotope. The equations governing this evolution can be written as:

$$\frac{dY_i}{dt} = \sum_{k \neq i} \lambda_{k \rightarrow i} Y_k + \sum_{k \neq i, l \geq k} [kl \rightarrow i] Y_k Y_l - \lambda_i Y_i - \sum_j [ij] Y_i Y_j \tag{3.1}$$

where $Y_i = \frac{X_i}{A_i}$ is the mole fraction (with X_i being the mass fraction of particle i), $\lambda_{k \rightarrow i}$ is the photodisintegration or $\beta-$decay rate of nucleus k leading to the formation of nucleus i, $[kl \rightarrow i]$ is the reaction rate between species k and l leading to the formation of nucleus i, $[kl \rightarrow i] = N_A \rho \langle \sigma v \rangle_{k,l \rightarrow i}$ (with N_A being the Avogadro number, ρ the density, and $\langle \sigma v \rangle_{k,l \rightarrow i}$ the Maxwellian-averaged product of the cross section and the velocity of the two nuclides k and l), λ_i is the total rate for all photodisintegration or $\beta-$decay channels of nucleus i, and $[ij]$ is the total rate for all exit channels involving destruction of nucleus i.

3.2.3. Numerical Treatment of Nuclear Abundances

The numerical treatment (in the sequential, 1-node execution) of the nuclear abundances for the whole set of isotopes included in our network is quite complex due to the large number of reactions that link a given isotope with the rest. To derive the new chemical composition of the whole envelope at a given time, we have to solve the system of differential equations given by equation (3.1). This can be written as a matrix equation, after linearization of the abovementioned system of equations (see Wagoner 1969):

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{X}_0 \tag{3.2}$$

where $\mathbf{A}$ is a matrix containing information on the different nuclear reaction rates, $\mathbf{X}$ is the matrix with the (unknown) new abundances, and $\mathbf{X_0}$ is the matrix containing the set of abundances of the previous step.

This equation is solved by means of an iterative technique, based on Wagoner's two-step linearization procedure (1969), as described in Prantzos *et al.* (1987). The procedure assumes

$\mathbf{X_0}$ as an initial guess to the new value of $\mathbf{X}$, and a first-order correction $\boldsymbol{\delta}^1\mathbf{X}$ to the initial $\mathbf{X_0}$ value is obtained applying a pseudo-Gaussian elimination technique to the equation

$$\mathbf{A}\cdot\delta^1\mathbf{X} = \mathbf{X}_0 - \mathbf{A}\cdot\mathbf{X}_0\,. \tag{3.3}$$

From this, a first-order approximation to the value of $\mathbf{X}$ is found:

$$\mathbf{X} \approx \mathbf{X}_1 = \mathbf{X}_0 + \boldsymbol{\delta}^1\mathbf{X}\,. \tag{3.4}$$

To achieve better accuracy, a second order correction $\boldsymbol{\delta}^2\mathbf{X}$ is obtained through a similar procedure:

$$\mathbf{A}\cdot\boldsymbol{\delta}^2\mathbf{X} = \mathbf{X_0} - \mathbf{A}\cdot\mathbf{X_1}\,, \tag{3.5}$$

leading to the final solution:

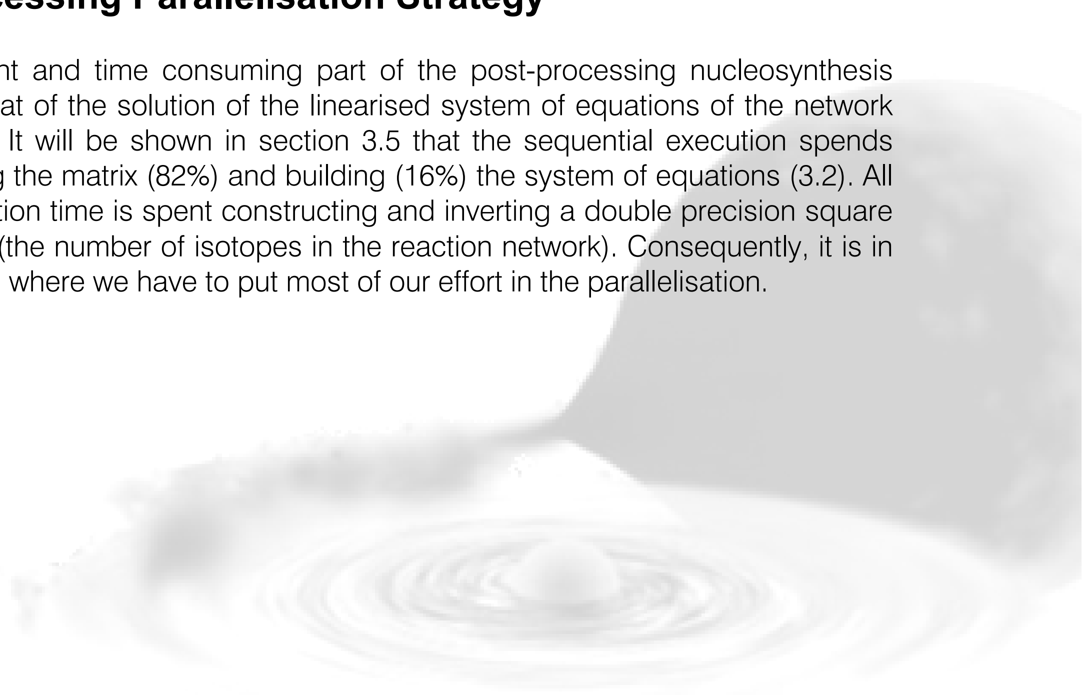
$$\mathbf{X} \approx \mathbf{X_2} = \mathbf{X_0} + \boldsymbol{\delta}^1\mathbf{X} + \boldsymbol{\delta}^2\mathbf{X} \tag{3.6}$$

which ensures conservation of the baryonic number up to 11 digits. This procedure is particularly suited for the special properties of matrix $\mathbf{A}$: essentially, a sparse matrix consisting of an upper left square matrix, an upper horizontal band, a left vertical band, and a diagonal band. This special geometry is due to the fact that the isotopes, ordered in terms of increasing atomic number, are only linked -through nuclear processes- either with close neighbours or with light particles (p, α, etc.).

Note that this is a sequential, 1-node algorithm that will not be used in the *parallel* numerical treatment of the nuclear abundances matrix. A method of parallel multifrontal decomposition will be used to solve the system of equations in a cluster of distributed-memory workstations (see section 3.3).

3.3. Post-Processing Parallelisation Strategy

The most important and time consuming part of the post-processing nucleosynthesis computation is that of the solution of the linearised system of equations of the network abundances derivatives. It will be shown in section 3.5 that the sequential execution spends most of the time inverting the matrix (82%) and building (16%) the system of equations (3.2). All in all, 98 % of the simulation time is spent constructing and inverting a double precision square matrix ($\mathbf{A}$) of order 606 (the number of isotopes in the reaction network). Consequently, it is in this part of the simulation where we have to put most of our effort in the parallelisation.



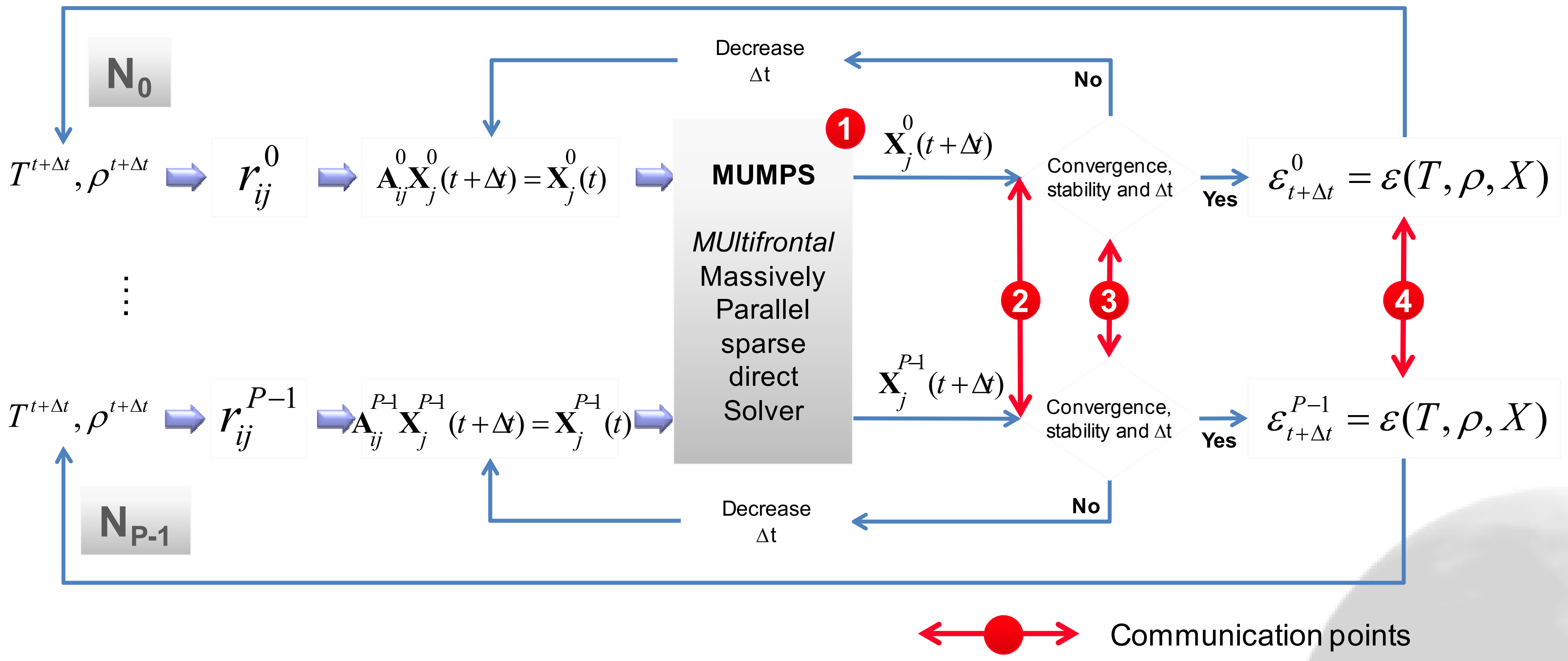

N0
$T^{t+\Delta t}, \rho^{t+\Delta t}$
r_{ij}^{0}
$\mathbf{A}_{ij}^{0}\mathbf{X}_{j}^{0}(t+\Delta t)=\mathbf{X}_{j}^{0}(t)$
Decrease Δt
No
MUMPS
MUltifrontal Massively Parallel sparse direct Solver
1
$\mathbf{X}_{j}^{0}(t+\Delta t)$
Convergence, stability and Δt
Yes
$\varepsilon_{t+\Delta t}^{0}=\varepsilon(T,\rho,X)$
2
3
4
$\mathbf{X}_{j}^{P-1}(t+\Delta t)$
Convergence, stability and Δt
Yes
$\varepsilon_{t+\Delta t}^{P-1}=\varepsilon(T,\rho,X)$
$T^{t+\Delta t}, \rho^{t+\Delta t}$
r_{ij}^{P-1}
$\mathbf{A}_{ij}^{P-1}\mathbf{X}_{j}^{P-1}(t+\Delta t)=\mathbf{X}_{j}^{P-1}(t)$
Decrease Δt
No
NP-1
Communication points

Fig. 18 Post-processing parallelisation strategy and processing stages

It is shown in Fig. 17 that the post-processing code consists of the following main processing stages:

1. Parallel Initialisation: Initialisation of parallel structures needed for the parallel computation.
2. Interpolation of Reaction Rates: Interpolate reaction rates at current temperature and density.
3. Matrix Assembly: Build the system of time-dependent equations of nuclides abundance.
4. Solution of the System of Equations: Solve the system of equations. Find new nuclides abundance.
5. Convergence and Stability: Check convergence, stability and Δt.
6. Released Energy Computation: Compute energy generation at $t+\Delta t$.

The system of equations is solved using MUMPS (Amestoy *et al.* 2001a, and 2006, MUMPS 2001). MUMPS stands for MUltifrontal Massively Parallel sparse direct Solver, and it is a widely used software application for the solution of large sparse systems of linear algebraic equations $Ax = b$ on distributed memory parallel computers. It represents one of the scarce professional and supported public domain implementations of the multifrontal method, and supports the solution of large linear systems with symmetric positive definite matrices, general symmetric matrices, and general non-symmetric matrices.

As stages 3 and 4 are by far the most time consuming part of the simulation, all the parallelisation strategy has been focused on providing the most efficient partitioning of the matrix $\mathbf{A}$, as required by the parallel solution of the system of equations performed by the parallel solver. Detailed information on the specific partitioning and distribution of the parallel construction of the matrix can be found in section 3.3.3. Fig. 17 shows the main parallelisation strategy. Reaction rates, construction of the matrix, stability check and energy generation are distributed amongst processors in a perfect parallel approach (see section 2.3.2). The processors communicate at four specific points during the simulation:

1. Communication during the parallel solution of the system of equations (MUMPS).
2. Communication once the system of equations is solved, because the solution is kept distributed (that is, every node holds only a portion of the solution vector), and therefore the distributed solution has to be shared amongst all processors. This is necessary to proceed with the second iteration of the Wagoner's two-step linearisation procedure (Wagoner, 1969), where the processors need the *complete* solution vector to build the new matrix $\mathbf{A}$ for the new iteration.
3. Communication to check convergence and stability of the solution and see whether the simulation is permitted to proceed or if the Δt has to be decreased.
4. Communication to sum up energy contributions from the distributed reactions (every node computes only the energy released by a subset of the reactions). This is tricky, because nodes only have available those reactions that are relevant to them for the distributed construction of matrix $\mathbf{A}$. As the number of reactions cannot be evenly distributed amongst processors, a specific procedure has been devised to even out (in a probabilistic way) the work load amongst nodes in the computation of the energy released.

In the following sections every processing stage is analysed in detail.

3.3.1. Parallel Initialisation

The initialisation of a parallel program is an essential part of the code; it is where all parallel structures and data partitioning strategy are defined, and where the *root* process (Process 0) reads in all the configuration parameters from the simulation configuration file, sending afterwards the relevant information to the rest of the processors. Amongst other parameters, process root broadcasts information such as the number of reactions (NRE), the number of isotopes (NIS), number of temperature grid points (NGRID), etc. Other data broadcasted are the number of points in the temperature-density profile, the temperature-density profile itself, the nuclear reaction network (isotopes involved in the reactions together with their Q value), the initial chemical composition, and parameters to configure the solution, stability and convergence of the system of linear equations.

Once the basic initialisation has taken place, it is necessary to set up the data structures that will allow the processors to do their parallel computations at later stages of the simulation:

- Compute input data needed by the parallel MUMPS solver, and the parallel processors.
- Set-up the distributed data local to each processor to build up the distributed assembled matrix. We provide the structure of the matrix on the host (root) at analysis, and MUMPS returns a mapping that we use afterwards to provide the matrix distributed according to the mapping on entry to the numerical factorization phase (see MUMPS user guide document 4.9.2.)
- Also, it is computed the number of reactions and *reaction pairs* that are relevant to the current process, in the construction of the distributed matrix $\mathbf{A}$.

Further details about the above data structures are given in the dedicated sections below.

3.3.2. Interpolation of Reaction Rates

All nodes have available the complete temperature grid points for all nuclear reactions, as well as the complete reaction network. The key point here is that every node only performs the interpolation of those reaction rates that it will be using afterwards in the construction of their local partition of the matrix $\mathbf{A}$. Given a matrix entry $\mathbf{A}(i, j)$, the node owner of this matrix entry requires all nuclear reactions that have a partial contribution to that specific element (see section 3.3.3 for details). Consequently, all nuclear reactions where elements i and j are involved will be relevant for the node owner of matrix element $\mathbf{A}(i, j)$.

Due to the fact that several nuclear reactions might contribute to a single matrix entry, it will not be possible to equally distribute the nuclear reactions amongst processors. For instance, in a parallel execution with four nodes, 529 reactions are relevant to Process 0, 2389 reactions are relevant to Process 1, 1469 reactions are relevant to Process 2, and 1053 reactions are relevant to Process 3. This is depicted in Fig. 19, where the relevant reactions are shown as black pixels in a two dimensional layout (53 x 67 = 3551) to ease the visualisation. Given a nuclear reaction $i(j,k)l$ there might be 8 possible reaction contributions pairs to the matrix $\mathbf{A}$: $\mathbf{A}(i,i)$, $\mathbf{A}(i, j)$, $\mathbf{A}(j, j)$, $\mathbf{A}(j,i)$, $\mathbf{A}(k,i)$, $\mathbf{A}(k, j)$, $\mathbf{A}(l,i)$, and $\mathbf{A}(l, j)$, according to the linearisation of Wagoner (1969). Thus, the specific partition of the matrix elements amongst processors is definitely the major determinant of the number of reactions that are relevant to each process.

It is evident that not all nodes get the same number of reactions, and additionally there is some overlap as some of the reactions are relevant to several processors at the same time. Nonetheless, the parallel execution of the interpolation of reaction rates yields significant improvements in performance compared to the sequential execution, where all 3551 reactions had to be interpolated by a single processor. For instance, in a parallel execution with four nodes, the reaction rates are interpolated (in average) 2.5 times faster than the sequential execution with a single node (see section 3.5 for further information).

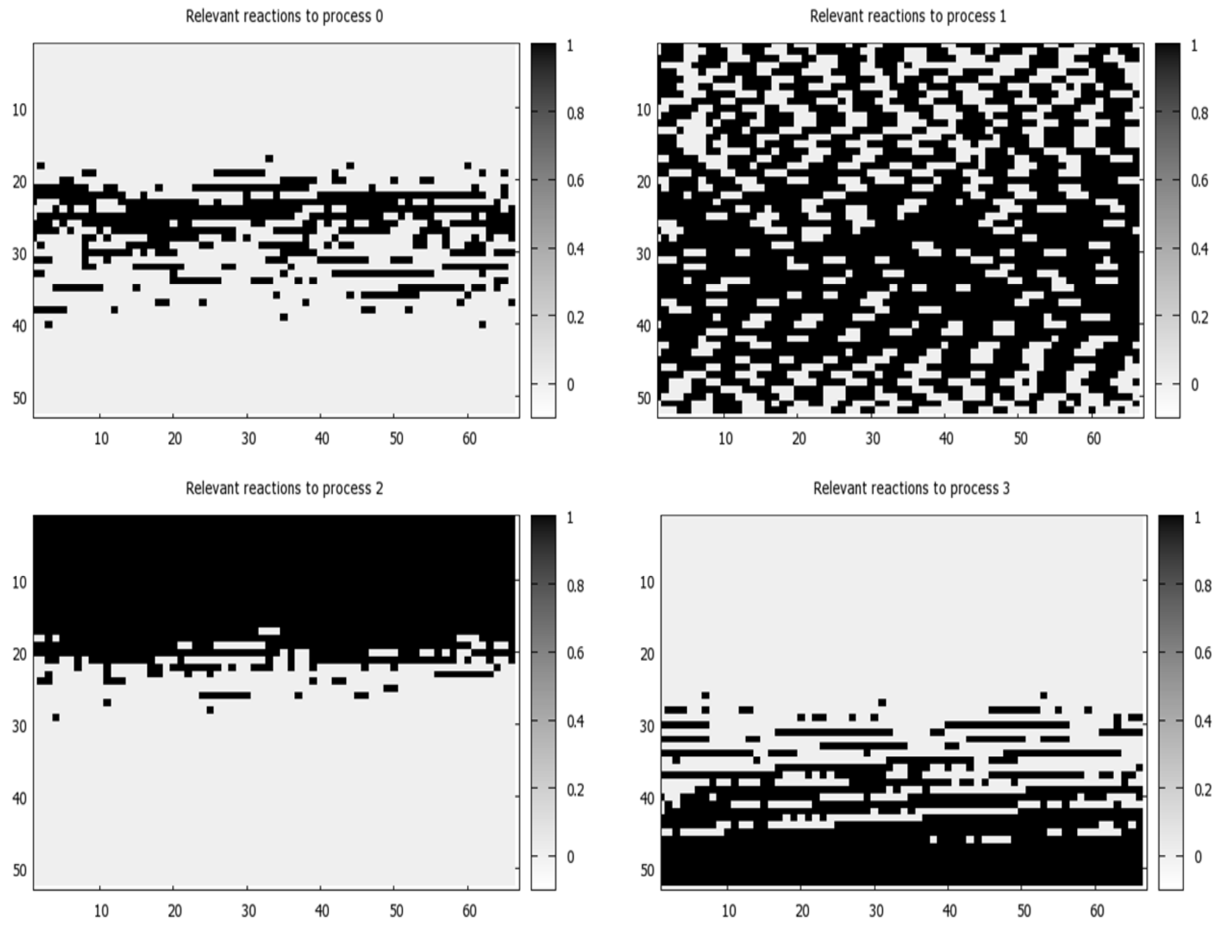


Fig. 19 Nuclear reactions partitioning in a parallel execution with 4 processors

At this processing stage, each node interpolates all their relevant reaction rates, in order to find the most accurate value given the current temperature and density. In order to do this, suppose that the current temperature (T_{curr}) falls between two temperature grid points, T_1 and T_2 (where $T_2 > T_{curr} > T_1$). We may define:

$$h = T_2 - T_1; \qquad b = \frac{T_{curr} - T_1}{h}. \tag{3.7}$$

From h and b the interpolation is carried out in two steps. First the interpolation is realised in the logarithmic scale, so that a logarithm of the interpolated reaction rate is obtained:

$$V_{\log} = \log 10(T_1) + b\left(\log 10(T_2) - \log 10(T_1)\right). \tag{3.8}$$

Secondly, the logarithmic interpolated reaction rate $V_{\log}$ is transformed back to linear in order to obtain the final interpolated reaction rate[4]:

$$V = 10^{V_{\log}}. \tag{3.9}$$

This is a costly operation because on the one hand it makes use of the FORTRAN's $\log 10$ and exponential 10^X intrinsic functions, which are rather computationally expensive (Mahaffy, 1997), and on the other hand this interpolation requires two computing steps.

An optimisation is proposed as part of this Thesis, aimed at improving the computational efficiency of this calculation. It can be shown that the interpolation can be performed in a single, linear step:

$$\begin{aligned} V = 10^{V_{\log}} &= 10^{\log 10(T_1) + b\left(\log 10(T_2) - \log 10(T_1)\right)} = \\ &= \frac{10^{\log 10(T_1)}\left(10^{\log 10(T_2)}\right)^b}{\left(10^{\log 10(T_1)}\right)^b} = \frac{T_1 \cdot T_2{}^b}{T_1^b} = T_2{}^b \cdot T_1^{1-b} \end{aligned} \tag{3.10}$$

Note the use of $V = T_2{}^b \cdot T_1^{1-b}$ instead of $V = \frac{T_2{}^b}{T_1^{b-1}}$, as FORTRAN multiplications are significantly faster than divisions.

Even though the relative computational gain will be small for a single operation, the above calculation is executed NRE times per iteration (3551 reactions in our network), with approximately 50k iterations per simulation (depending on the model). The total amount works out at roughly $177 \cdot 10^6$ reaction rates interpolations per simulation, which might provide for significant savings at the end of the computation.

3.3.3. Matrix Assembly

One of the most time consuming stages of the computation is the construction of the matrix $\mathbf{A}$ that arises from the linearisation of the set of differential equations describing the temporal evolution of the network abundances:

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{X}_0. \tag{3.11}$$

[4] Note that for the sake of conciseness, we have omitted the corrective factors ρ / μ_e for electron capture, and ρ^{n-1} for n-particle reactions.

As shown in Fig. 20, matrix $\mathbf{A}$ is a sparse matrix with a rather simple geometry, as long as the isotopes are ordered in terms of increasing atomic mass (Prantzos *et al.* 1987). This is owing to the fact that isotopes are coupled only by light particle reactions. This structure can be described in terms of an upper left square matrix, an upper horizontal band, a left vertical band, and a diagonal band.

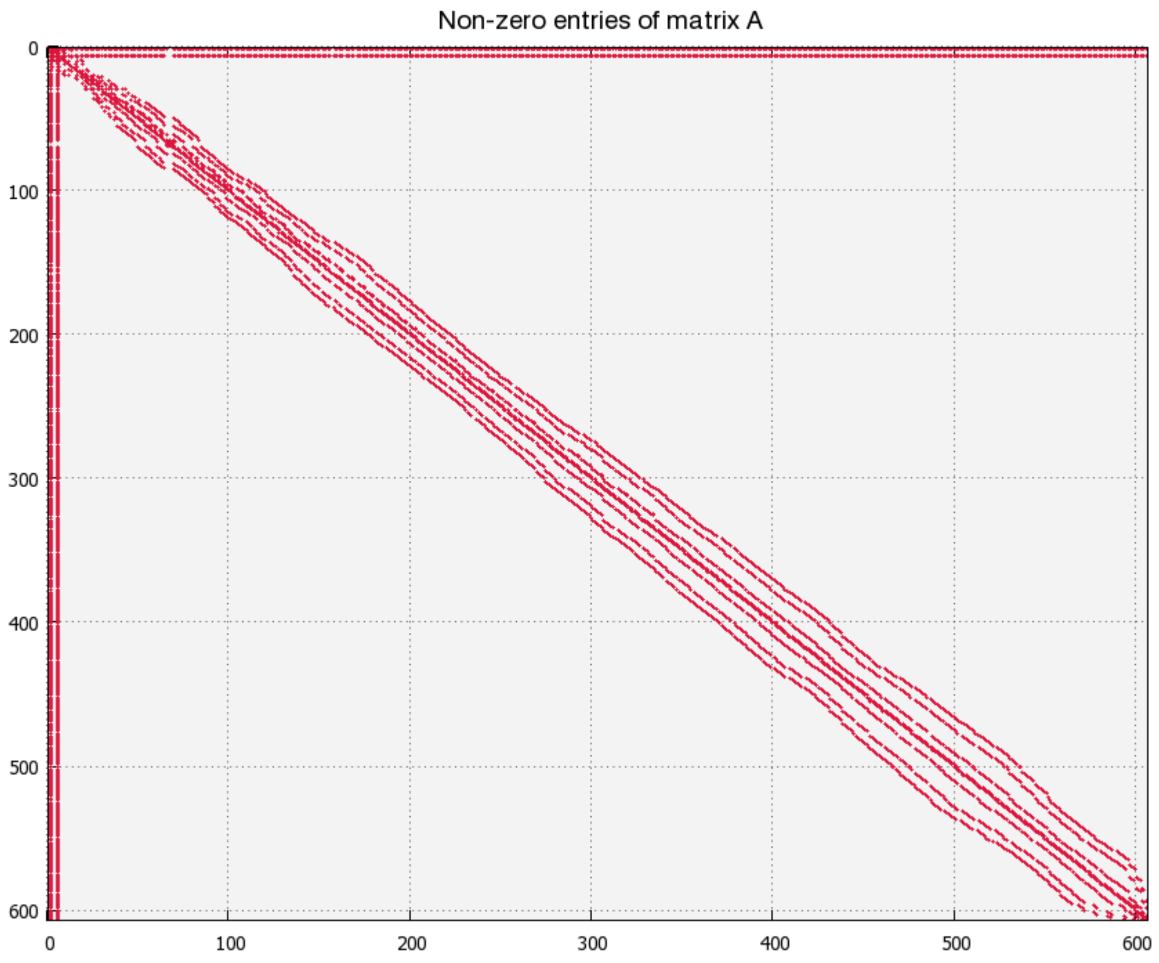


Fig. 20 Non-zero entries of matrix A

Matrix $\mathbf{A}$ contains 368,449 elements (607 elements squared matrix), out of which only 6,454 elements are non-zero (that is, the nuclear reactions only contribute to 6,454 entries of the above matrix). It is therefore a very high sparseness factor; this fact will have to be taken into account in the distributed construction of the matrix.

Given a nuclear reaction $i(j,k)l$ there might be 8 possible reaction contributions pairs to the matrix $\mathbf{A}$: $\mathbf{A}(i,i)$, $\mathbf{A}(i,j)$, $\mathbf{A}(j,j)$, $\mathbf{A}(j,i)$, $\mathbf{A}(k,i)$, $\mathbf{A}(k,j)$, $\mathbf{A}(l,i)$, and $\mathbf{A}(l,j)$:

$$reaction \quad i(j,k)l \xrightarrow{Contributes} \mathbf{A} = \begin{pmatrix} \ddots & & & \iddots \\ & A_{ii} & \cdots & A_{ij} & \\ & \vdots & & \vdots & \\ & A_{ji} & \cdots & A_{jj} & \\ & \vdots & & \vdots & \\ & A_{ki} & \cdots & A_{kj} & \\ & \vdots & & \vdots & \\ & A_{li} & \cdots & A_{lj} & \\ \iddots & & & \ddots \end{pmatrix}$$

where A_{ii}, A_{ij}, …, A_{lj} are the *partial contributions* of the reaction $i(j,k)l$ to the matrix elements $\mathbf{A}(i,i)$, $\mathbf{A}(i,j)$, …, $\mathbf{A}(l,j)$. The matrix elements are, according to the linearisation of Wagoner (1969):

$$\mathbf{A}(i,i) = 1 + \sum_{i,j,k} \left(\frac{\Delta t \cdot N_i \cdot [ij]_k}{N_i! \cdot N_j! \cdot (N_i + N_j)} N_i \cdot Y_i^{N_i - 1} \cdot Y_j^{N_j} \right)$$

$$\mathbf{A}(j,j) = 1 + \sum_{i,j,k} \left(\frac{\Delta t \cdot N_j \cdot [ij]_k}{N_i! \cdot N_j! \cdot (N_i + N_j)} N_j \cdot Y_j^{N_j - 1} \cdot Y_i^{N_i} \right)$$

$$\mathbf{A}(i,j) = \sum_{i,j,k} \left(\frac{\Delta t \cdot N_i \cdot [ij]_k}{N_i! \cdot N_j! \cdot (N_i + N_j)} N_j \cdot Y_j^{N_j - 1} \cdot Y_i^{N_i} \right)$$

$$\mathbf{A}(j,i) = \sum_{i,j,k} \left(\frac{\Delta t \cdot N_j \cdot [ij]_k}{N_i! \cdot N_j! \cdot (N_i + N_j)} N_i \cdot Y_i^{N_i - 1} \cdot Y_j^{N_j} \right)$$

$$\mathbf{A}(k,i) = -\sum_{i,j,k} \left(\frac{\Delta t \cdot N_k \cdot [lk]_j}{N_k! \cdot N_l! \cdot (N_k + N_l)} N_i \cdot Y_i^{N_i - 1} \cdot Y_j^{N_j} \right)$$

$$\mathbf{A}(k,j) = -\sum_{i,j,k} \left(\frac{\Delta t \cdot N_k \cdot [lk]_j}{N_k! \cdot N_l! \cdot (N_k + N_l)} N_j \cdot Y_j^{N_j - 1} \cdot Y_i^{N_i} \right)$$

$$\mathbf{A}(l,i) = -\sum_{i,j,k} \left(\frac{\Delta t \cdot N_l \cdot [lk]_j}{N_k! \cdot N_l! \cdot (N_k + N_l)} N_i \cdot Y_i^{N_i - 1} \cdot Y_j^{N_j} \right)$$

$$\mathbf{A}(l,j) = -\sum_{i,j,k}\left(\frac{\Delta t \cdot N_l \cdot [lk]_j}{N_k! \cdot N_l! \cdot (N_k + N_l)} N_j \cdot Y_j^{N_j - 1} \cdot Y_i^{N_i}\right) \tag{3.12}$$

At initialisation time, MUMPS is provided with the sparseness pattern of the matrix describing the temporal evolution of the network abundances (Fig. 20). The algorithm automatically analyses the structure of the matrix on the host (root process), and returns a mapping structure holding the optimised distribution of the matrix elements amongst the processors. Depending of the sparseness distribution and the number of processors, MUMPS computes the optimum distribution so as to minimise the factorisation and solution phases to be carried out afterwards. In turn, each processor builds, at initialisation time, a local data structure containing the coordinates of the matrix elements assigned to them, as well as the reactions and reaction *pairs* that they need to totally calculate a specific matrix entry. It is prior to the factorisation and solution phases that every processor will construct a local matrix $\mathbf{A\_loc}$ that will be used in the parallel numerical factorisation and solution phases. For instance, in Fig. 21 overleaf, it is presented the elements distribution amongst a group of 4 processors. Note that the optimum distribution is not an equitable split of the matrix, but certain processors have more elements in $\mathbf{A\_loc}$ than the others.

It is important to remark that there is no overlap between processors, that is, one matrix element is assigned to one and only one processor. Note, however that the split of the matrix does keep a pattern, having each processor hold matrix entries of a specific zone or area of the matrix. This fact is easily seen in Fig. 21, where it is clear that process 2 holds most of the elements in the top left area of the matrix, and process 3 holds most of the bottom right entries. Process 0 and process 1 hold specific portions of the diagonal, horizontal and vertical bands, in the mid section of matrix A. For processors 0 and 1, though, there is much more overlap in the areas where they have responsibility, and their boundaries are more diffuse.

The matrix $\mathbf{A\_loc}$ is stored in an *assembled format* (as opposed to *elemental format*). The following components of the structure define the distributed assembled matrix input, and are *local* to each of the processors (that is, the structures hold different values for every processor):

- $\mathrm{NZ\_loc}$: is the number of entries local to a processor. For instance, in the example shown in Fig. 21 $\mathrm{NZ\_loc} =$ 792 for process 0, $\mathrm{NZ\_loc} =$ 1181 for process 1, $\mathrm{NZ\_loc} =$ 2613 for process 2, and $\mathrm{NZ\_loc} =$ 1868 for process 3.
- $\mathbf{IRN\_loc, JCN\_loc}$: are integer arrays of length $\mathrm{NZ\_loc}$ containing the row and column indices, respectively, for the matrix entries.
- $\mathbf{A\_loc}$: is a double precision array of dimension $\mathrm{NZ\_loc}$. $\mathbf{A\_loc}(k)$ is set to the value in row $\mathbf{IRN\_loc}(k)$ and column $\mathbf{JCN\_loc}(k)$.

In the parallel construction of the matrix, each process traverses its own local structure holding the matrix entries and the relevant reaction rates (that have been interpolated locally, see section 3.3.2), and fills the elements of their local matrix $\mathbf{A\_loc}$ according to partial contributions (3.12). Significant improvements in computation time are achieved in the parallel construction of matrix $\mathbf{A}$ (see section 3.5). For instance, in a parallel execution with four nodes, the matrix is constructed (in average) 6.5 times faster than the traditional sequential execution with a single node. This is an extremely good speed-up accomplished in this phase of the computation.

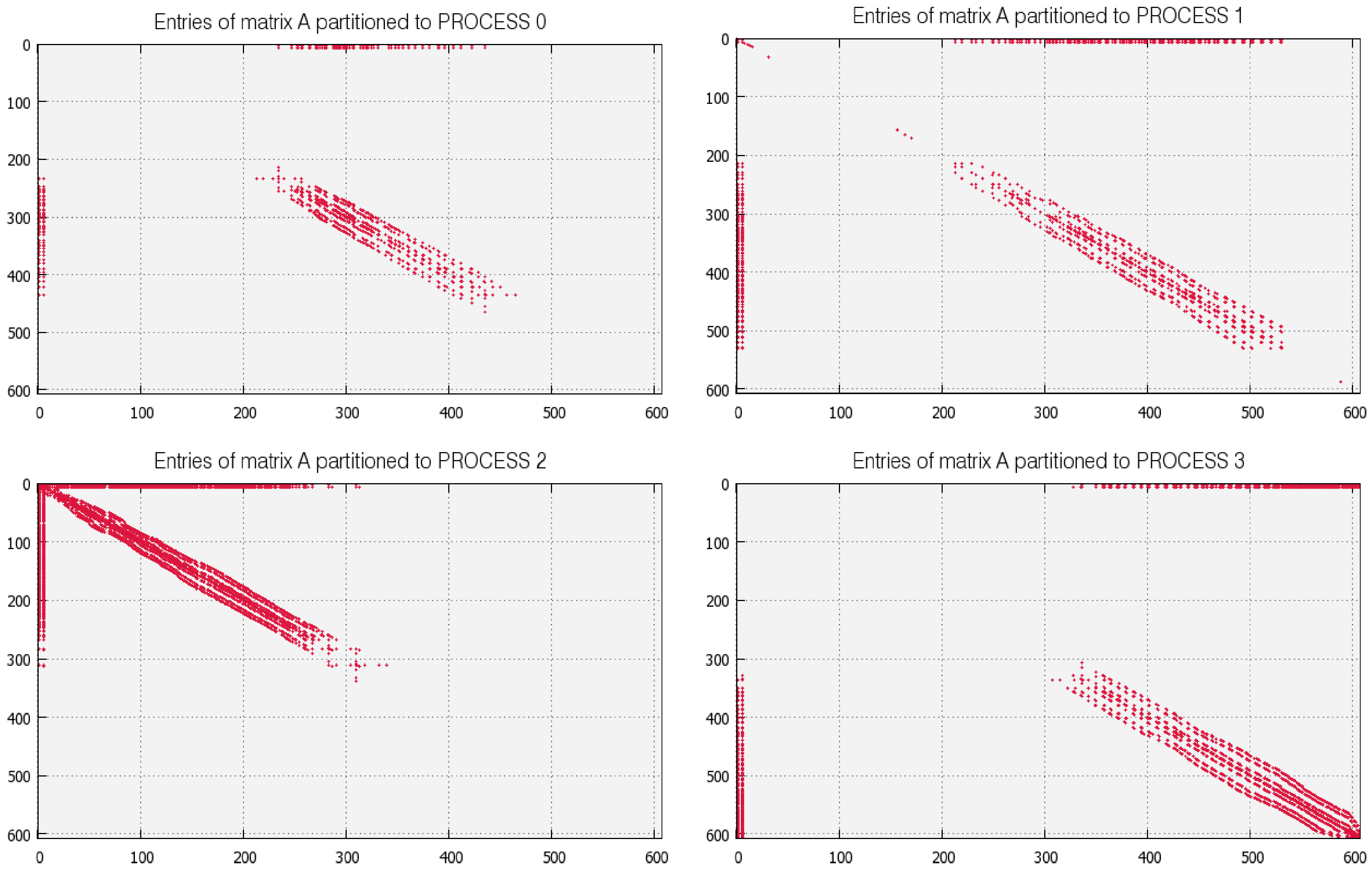


Fig. 21 Parallel distribution of matrix A with four processors

3.3.4. Solution of the System of Equations

At this stage of the computation, each of the processors has already made up the structures needed for the parallel solution of the system of equations using the MUMPS software. MUMPS requires that at the solution phase, **A_loc, IRN_loc, JCN_loc** must have been provided locally to each processor. It must be noted that the right hand side (RHS) is not distributed, but has to be provided centralised on the root process, this fact entails that the root process must have the complete solution vector with the mass fraction abundances available from the previous iteration.

The MUMPS package uses a multifrontal approach to factorize the matrix (Duff and Reid 1983). The principal feature of a multifrontal method is that the overall factorization is described (or driven) by an assembly tree (see Fig. 22, left).

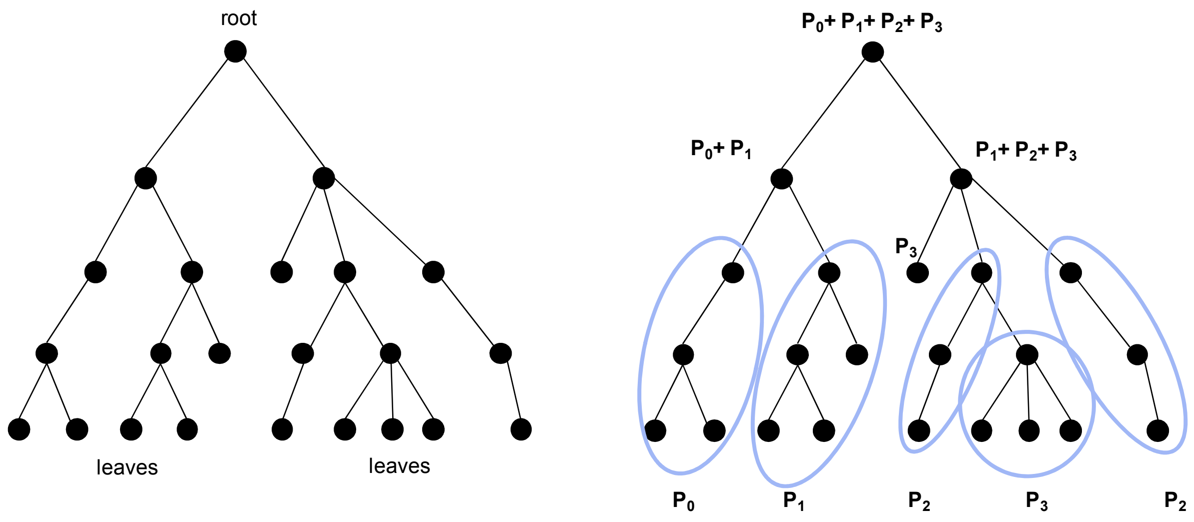


Fig. 22 Example assembly tree (left) and a possible distribution over four processors (right)[5]

At each node in the tree, one or more variables are eliminated using steps of Gaussian elimination on a dense matrix; the frontal matrix. Each edge in the tree represents the movement of data of a child node to its parent (which is the adjacent node in the direction of the root). An important aspect of the assembly tree is that it only defines a partial order for the factorization. That is, arithmetic operations at a pair of nodes, where neither lies on a path from the other to a root node, are independent. For example, work can commence in parallel on all the leaf nodes of the tree. Operations at the other nodes in the tree can proceed as soon as the data is available from the children of the node. There is thus good scope for exploiting parallelism, especially since assembly trees for practical problems contain many thousands of nodes. For nodes far from the root, to keep communication to a minimum while maintaining a high level of parallelism, MUMPS maps a complete sub tree onto a single processor of the target machine (see Fig. 22, right).

On some networks with low bandwidth, centralizing the solution on the host processor might be a costly part of the solution phase. As this is critical to the performance of the parallel

[5] Source: *MUMPS: A Multifrontal Massively Parallel Solver* by Patrick Amestoy, Iain Duff, Jacko Koster, and Jean-Yves L'Excellent. ERCIM News No.50, July 2002

application, in this Thesis we have decided for the solution to be left distributed over the processors[6] after the system of equations is solved (MUMPS also allows the user to configure the software so as to centralise the solution vector on the root processor once the system has been solved). In this regard, each of the processors holds a non overlapping subset of the elements of the solution vector (see Fig. 23), that is, a subset of the abundances for the new Δt. It must be taken into account that in this case, the solution must then be exploited in its distributed form; this fact requires that subsequent processing stages (e.g. stability and convergence) must be also parallelised (see section 3.3.5).

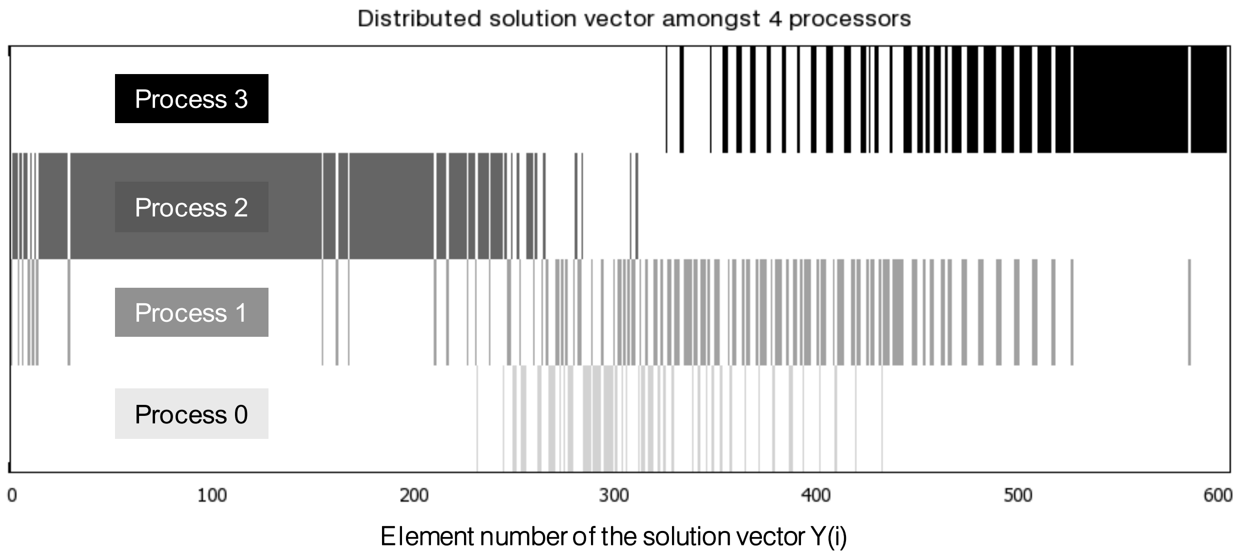


Fig. 23 Distributed solution vector with 4 processors

As with the distribution of matrix entries, the partition of the solution vector is not equitable; for instance in our parallel implementation of the post-processing code with four processors, process 0 keeps 58 elements of the solution vector, whereas process 1, 2 and 3 hold 140, 241, and 167 elements respectively. Note that there exists a correlation between the number of elements of the matrix $\mathbf{A}$ assigned to a specific processor and how many entries of the solution vector are stored locally for that processor (nodes with more matrix elements, have larger local solution vectors). Moreover, it is shown in Fig. 23 that also the location of the elements of the solution assigned to each node, has a correlation with the location of the matrix elements; process 2 and 3 hold most of the solution elements at the beginning and end of the solution vector (as it were with the matrix entries - see Fig. 21), whereas process 0 and 1 keep most of the solution elements around the middle indexes of the solution vector.

Unfortunately, it will be shown in section 3.5 that the parallel solution of the system of equations takes (in the general case) considerably longer to calculate as compared to the sequential execution. This disappointing result stems from the fact that the post-processing nucleosynthesis application at hand is a loosely coupled application, where the time needed for communications quickly dwarves not only the time spent in computation tasks, but only the

[6] Note, however, that as the two-step method of Wagoner (1969) consists of two iterations (in both of which the system of equations must be linearised and solved), it will be necessary to distribute the solution vector to all processors at the end of the first iteration (all to all communication), since it will be used in the construction of the matrix in the next iteration. It is only after the second iteration that the distributed solution can be exploited in the parallel analysis of convergence and stability.

time saving accomplished in the parallelisation of the other processing stages (interpolation of reaction rates and matrix assembly). We are hopelessly smack in the middle of the worst possible scenario for parallelisation; a loosely coupled application with a very small computing-to-communication time ratio.

3.3.5. Convergence and Stability

In the solution of the system of equations arising from the linearisation of the abundance derivatives, it is necessary to check for convergence and stability of the solution. Convergence and accuracy rely heavily on the chosen Δt; in this regard the variation of the mass fraction abundances has to be kept below a limit. It has been found helpful to limit the time step by changes in the chemical composition, assuming that the relative abundance variation of the most abundant nuclei (i.e., Y > 10^{-14}, with Y = X/A being the mole number) do not exceed 15% (Wagoner 1969), that is:

$$\frac{Y_i^{t+\Delta t} - Y_i^t}{Y_i^t} = \frac{\Delta Y_i}{Y_i^t} < 15\% \quad \forall\, Y_i > 10^{-14} \tag{3.13}$$

To determine the new Δt using the largest abundance variation, the complete solution vector obtained from the solution of the system of equations for the abundances derivatives, has to be traversed and searched for the largest abundance variation amongst the most important nuclei. If the largest abundance variation is too large, the results of the current step are cancelled and the time step reduced. A new system of equations is constructed and solved for the new time step, for which convergence and stability has also to be verified. It is worth mentioning that since the nucleosynthesis code is implicit, we do not need to deal with other restrictive conditions on the time step, such as the Courant- Friedrichs-Levy condition (Kippenhahn *et al.* 1996).

In the parallel execution of the test of stability, every processor holds a portion of the solution vector (see Fig. 23), and therefore all processors traverse their local solution vector looking for the largest abundance variation with respect to the previous solution vector. At this point there is no other alternative but to distribute all the largest variations amongst all processors (all-to-all communication), so that every processor knows whether any of the nuclei as changed beyond the maximum abundance variation. At this point there may be found two possible outcomes:

1. At least one relative abundance variation of the most abundant nuclei exceeds the imposed limit: In this case, all processors reduce the time step, and proceed to build and solve a new system of equations with the new time step.
2. All abundance variations are below the limit: In this case, all processors share their local, partial solution vectors in an all-to-all communication scheme, so that every processor obtains the complete solution vector. This is necessary, since the processors need the complete solution vector to construct their local portion of matrix $\mathbf{A}$ at the next iteration.

Note that since the several processors have to traverse only a subgroup of the nuclides, the check of stability is performed much faster in the parallel version as compared to the sequential code, where all nuclides' variations have to be analysed by a single processor. For instance, with a parallel execution of 4 processors, the stability check is executed almost 3 times faster than the sequential version (see section 3.5). It must be stressed, however, that this accounts only for the stability check itself, and it does not take into account the communication times of solution vector and maximum relative abundance variations distribution. On top of that, the stability check holds a rather small fraction of all computing time (0.06% in its sequential form), therefore the little benefits in the parallel execution will have little to none impact on the overall performance of the parallel execution.

3.3.6. Energy Released Computation

Once we have obtained how the different nuclear reaction rates have made the abundances evolve at $t + \Delta t$, the calculation of the energy generation rate at this specific time step of the computation can be obtained by summing the energy generated by all reactions in $t + \Delta t$. That is:

$$\varepsilon_{total} = \sum_{i=1,NRE} \varepsilon_i \qquad \left(\frac{erg}{g \cdot s} \right) \tag{3.14}$$

The parallelisation of this calculation might seem an easy task at first glance; that is, providing an equal number of reactions to each of the processors and then having each node calculate the partial nuclear energy generation rate for all the reactions assigned to it, which may afterwards be aggregated to the values obtained by all processors, therefore yielding the total energy generation rate.

However, a close scrutiny of the reaction rates that are available to each of the processors (see section 3.3.2), reveals that the problem is actually far more complicated than that. Firstly there is no such thing as an equitable distribution of reaction rates amongst processors (Fig. 19), and second, and more importantly, there exist reaction rates that are relevant to several processors at the same time, which would cause that some of the reactions were included more than once in the calculation of the nuclear energy generation rate, yielding wrong results for ε_{total}. Alternatively, if we have all processors keep all reaction rates available, we would loose the benefit of parallelisation of the interpolation of the reaction rates, which, in our parallelisation approach, is limited to the strictly necessary reaction rates in order for the processor to be able to construct its portion of matrix $\mathbf{A}$.

In order to overcome this hurdle, in this Thesis we have devised a probabilistic approach that provides a nearly optimum distribution of the reaction rates amongst processors, with no overlaps, and evening up (as much as possible) the partition of reaction rates for the parallel released energy computation. Let NRE be the number of nuclear reaction rates used in the simulation ($NRE = 3551$ in this Thesis), NRE_i_loc the relevant reactions[7] available to processor i, P the number of processors, $ENRE_i\_loc$ the number of reactions assigned to processor i for released nuclear energy calculation, therefore:

1. Sort processors in ascending order of NRE_loc.

[7] The term *relevant reaction* refers to those reaction rates that are needed for a specific processor for the construction of their parallel portion of matrix $\mathbf{A}$, and are therefore calculated by the processor during the parallel interpolation of reaction rates.

2. First assign all those processors with a number of relevant reactions below the average $NRE\_loc < NRE / P$. This avoids that processors with less relevant reactions suffer from 'starvation' of reactions.
3. While there are still un-assigned reactions:
    a. Pick-up randomly a nuclear reaction r
    b. If the reaction r has not been assigned to any other processor, assign r to the processor that:
        i. Has reaction r as a relevant reaction, and
        ii. Has the minimum $ENRE\_loc$.

Note the use of random numbers in the range [1:NRE] to achieve a more even distribution of reactions. If reactions were checked linearly, processors with big chunks of contiguous relevant reactions would get more reactions (hence more computing work) than the other processors.

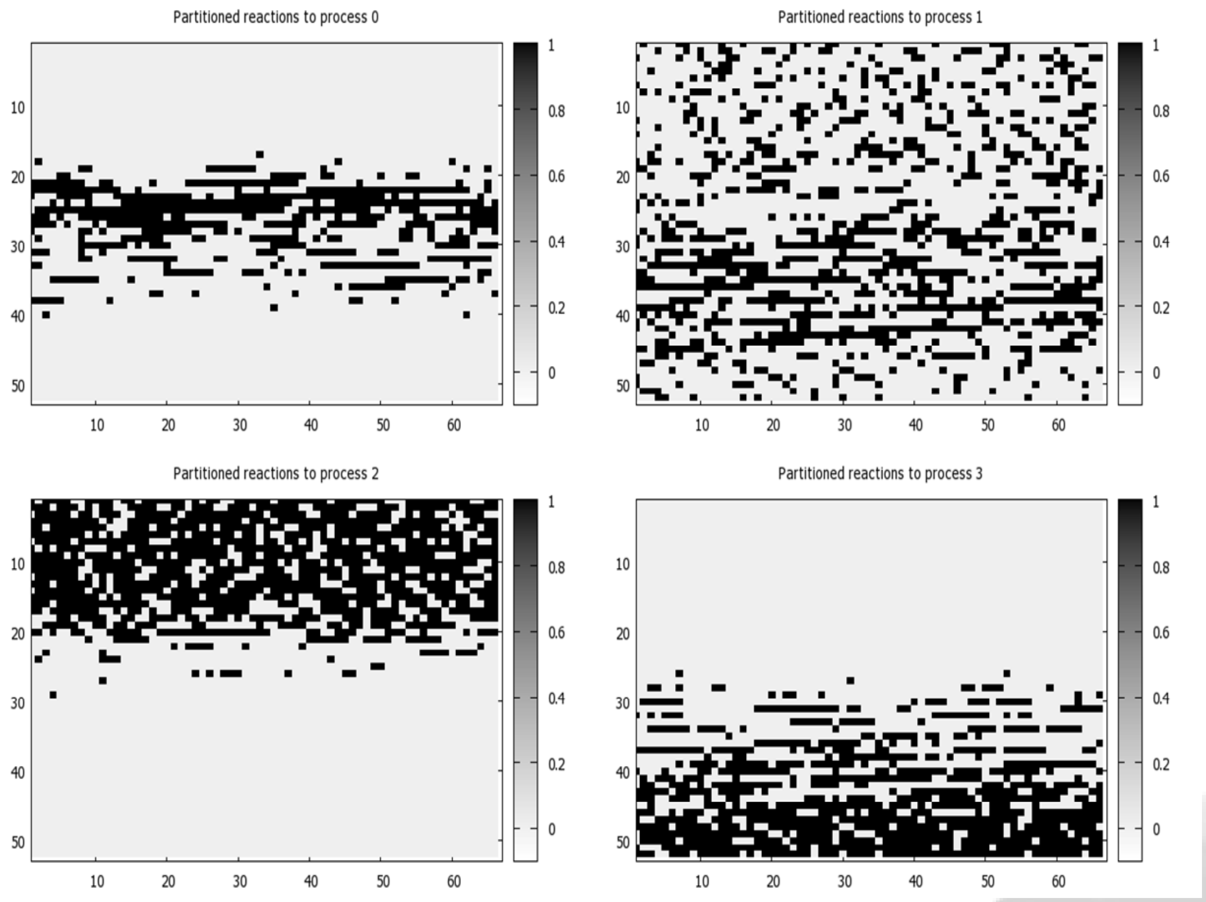


Fig. 24 Partition of nuclear reactions to 4 processors for parallel nuclear energy calculation

In Fig. 24 it is shown the partition of nuclear reactions to four processors for the parallel calculation of the released nuclear energy. Compare it with the relevant reactions distribution depicted in Fig. 19; the former has no overlapping reactions amongst processors, and the number of reactions assigned to each processor has been levelled out. This will provide for a more uniform distribution of the workload between nodes.

	Process 0	Process 1	Process 2	Process 3
Assigned reactions to calculate nuclear energy generation (ENRE_loc)	529	2389	1469	1053
Relevant reactions to construct matrix A (NRE_loc)	529	1079	1085	858

Table 1 Partition of nuclear reactions to four processors

Table 1 shows the precise values of NRE_i_loc and $ENRE_i\_loc$ for the parallel execution with four processors depicted in Fig. 24. It can be seen that those processors with less reaction rates available than the average retain all their available reaction rates to calculate their contribution to the energy generation (i.e. Process 0), whereas other processors with many more reaction rates available (i.e. Process 1), reduce considerably the number of reactions that have to take into account in their partial calculation of the released energy.

3.4. Validation of the Parallel Application

A parallel application has to be validated both in performance and in the correctness of their output results. There is no point in getting results much faster if at the end of the day the parallel application yields biased, wrong or incomplete results. In this regard, the baseline program provides a built-in mechanism for validating the results of the parallel program (it has to yield the same results as the sequential code for all simulation inputs), as well as a basis for calculating improvements in the performance (that is, how much faster is the parallel version with regard to the sequential version).

The validation has been realised on the basis of the following: 1) Abundance evolution for the most abundant nuclei (i.e. those with final abundance $X_i > 10^4$), 2) Error in the solution of the system of equations, and 3) Released Energy Calculation. These are described in the following.

Fig. 25, Fig. 26, and Fig. 27 show accuracy comparison between the sequential and parallel solver, in the solution of the system of equations

$$\mathbf{A}_t \cdot \mathbf{X}_{t+\Delta t} = \mathbf{X}_t \,. \tag{3.15}$$

Accuracy of the final solution is defined is terms of the Mean Absolute Error (MAE), the Mean Square Error (MSE), and the Mean Relative Error (MRE), which we define as the average error across all NIS (number of isotopes, NIS = 606 in our simulation) elements of the solution vector:

$$MAE = \frac{1}{NIS} \sum_{i,j} \left(X_t(i) - A_{ij} X_{t+\Delta t}(i) \right) \tag{3.16}$$

$$MSE = \frac{1}{NIS} \sum_{i,j} \left(X_t(i) - A_{ij} X_{t+\Delta t}(i) \right)^2 \tag{3.17}$$

$$MRE = \frac{1}{NIS} \sum_{i,j} \left(\frac{X_t(i) - A_{ij} X_{t+\Delta t}(i)}{X_t(i)} \right). \tag{3.18}$$

The solution of the system has to be provided with sufficient precision, in order to provide accurately the net flows of the nuclear reactions, both at high temperatures (when some nuclear reactions proceed in both directions almost equally rapid), and also for those small net flows resulting from reactions in equilibrium (Wagoner 1969). On top of that, it is important to conserve the baryonic number, so that numerical errors are not propagated after a large number of iterations (Prantzos *et al.* 1987).

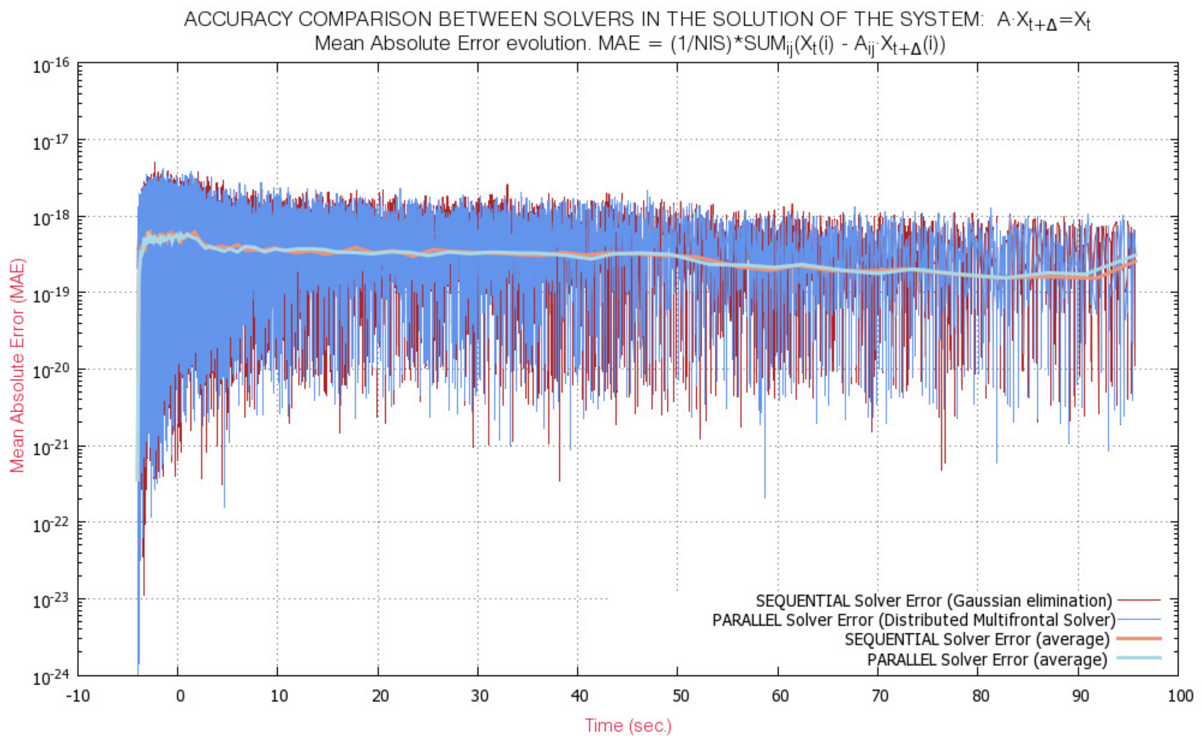


Fig. 25 Mean absolute error comparison between sequential and parallel solvers

Fig. 25 and Fig. 26 demonstrate that both solvers (the sequential solver using a pseudo-Gaussian elimination technique and the parallel solver using a parallel multifrontal decomposition algorithm) achieve precisely the same level of accuracy in the solution of the system of equations. It is shown that the temporal evolution of the error, suffers larger fluctuations during the first stages of the simulation, only to get slightly more stable as the computation proceeds. Note however the great variability of the error during the entire time-span of the simulation. This behaviour is generalised across sequential and parallel solvers. In turn, Fig. 27 shows the mean relative error. It is representative to compare both errors, and see how the several patterns are common between both solvers. Even though the smoothed line (included as an aid for the average visualisation), fluctuates differently for the parallel and sequential solvers, both are almost equivalent in terms of the average relative error obtained. Differences in the behaviour are associated to the different algorithms used in the solution of the system of equations with the temporal evolution of the abundances derivates.

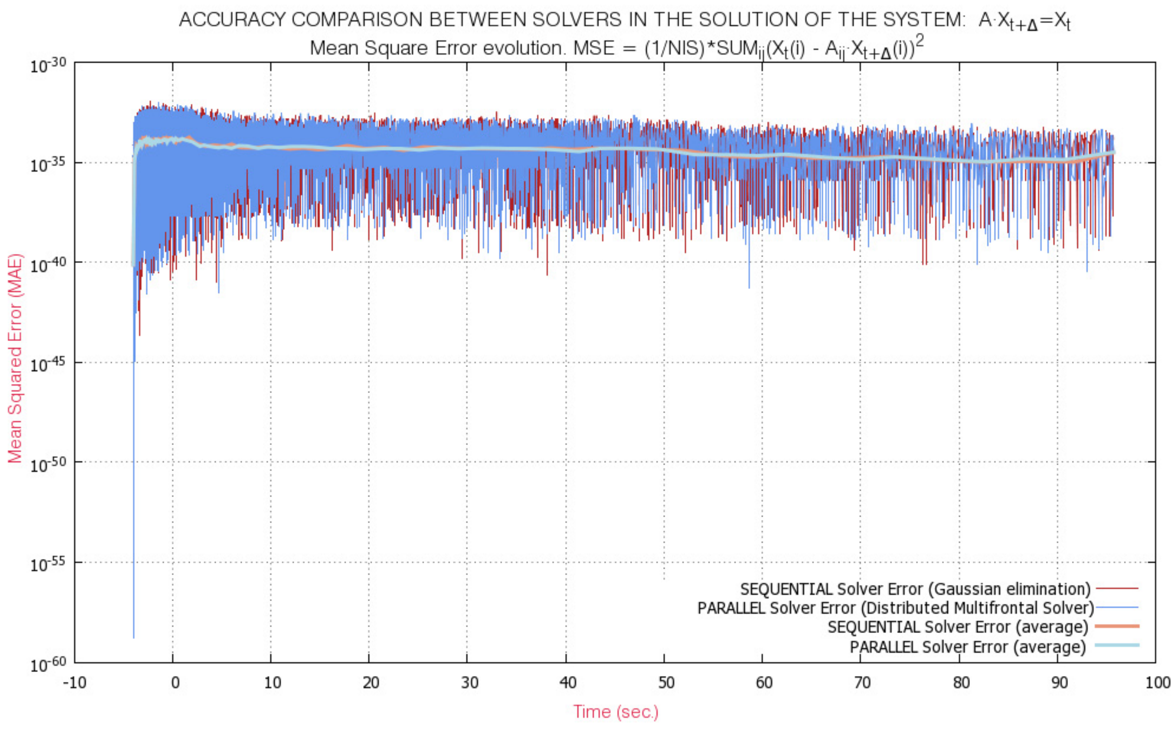


Fig. 26 Mean squared error comparison between sequential and parallel solvers

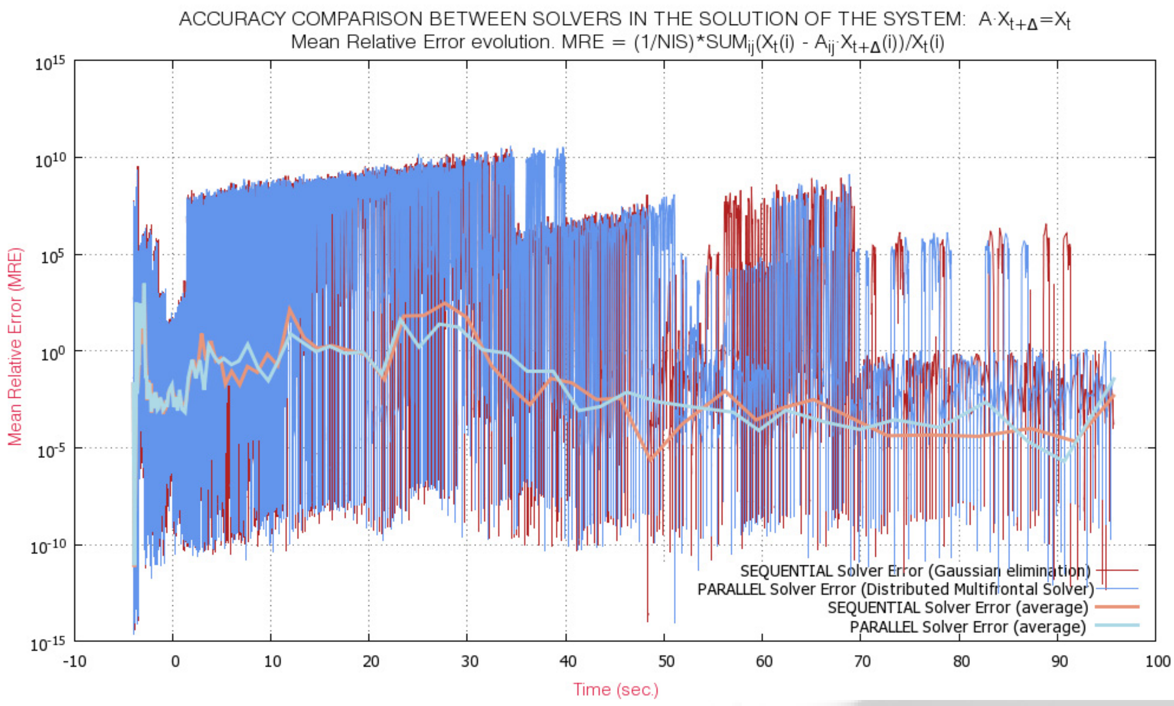


Fig. 27 Mean relative error comparison between sequential and parallel solvers

A second and essential element of comparison between the solution of the sequential and parallel versions of the nucleosynthesis post-processing code, is the time evolution of the abundances. It is clear that the results of the parallel application shall be within controlled bounds with respect to the sequential, *baseline* results. Again, comparison has been made for those nuclei with final abundances above $X_i > 10^4$.

Nuclei	Final abundance SEQ	Final abundance PAR	Nuclei	Final abundance SEQ	Final abundance PAR
H 1	1,99E-01	1,97E-01	MO 86	1,53E-02	1,58E-02
HE 4	2,13E-02	2,11E-02	NB 87	2,26E-03	2,42E-03
ZN 60	7,04E-03	6,98E-03	MO 87	1,02E-02	1,05E-02
GE 64	7,21E-02	7,06E-02	TC 87	2,04E-03	2,00E-03
SE 68	2,04E-01	2,00E-01	MO 88	6,08E-03	6,41E-03
SE 69	1,04E-03	1,08E-03	TC 88	3,72E-03	3,74E-03
BR 71	1,26E-03	1,32E-03	MO 89	2,91E-03	3,10E-03
KR 72	1,31E-01	1,29E-01	TC 89	7,74E-03	7,92E-03
KR 73	1,97E-03	2,09E-03	MO 90	2,31E-03	2,50E-03
KR 74	1,19E-03	1,28E-03	TC 90	4,34E-03	4,51E-03
RB 75	2,82E-03	2,96E-03	RU 90	6,58E-03	6,65E-03
RB 76	2,48E-03	2,65E-03	TC 91	5,65E-03	6,02E-03
SR 76	7,13E-02	7,07E-02	RU 91	5,06E-03	5,14E-03
SR 77	3,57E-03	3,78E-03	RU 92	8,63E-03	9,15E-03
SR 78	4,16E-03	4,38E-03	RH 92	1,14E-03	1,13E-03
Y 79	4,35E-03	4,52E-03	RU 93	3,68E-03	3,95E-03
Y 80	7,37E-03	7,79E-03	RH 93	5,48E-03	5,67E-03
ZR 80	3,38E-02	3,34E-02	RH 94	6,39E-03	6,80E-03
Y 81	1,21E-03	1,32E-03	PD 94	2,69E-03	2,73E-03
ZR 81	7,43E-03	7,65E-03	RH 95	4,30E-03	4,66E-03
ZR 82	2,18E-02	2,23E-02	PD 95	4,30E-03	4,47E-03
ZR 83	6,45E-03	6,85E-03	PD 96	1,04E-02	1,12E-02
NB 83	8,41E-03	8,43E-03	PD 97	1,56E-03	1,70E-03
ZR 84	3,31E-03	3,58E-03	AG 97	1,94E-03	2,03E-03
NB 84	1,19E-02	1,23E-02	AG 98	2,85E-03	3,07E-03
ZR 85	9,03E-04	9,91E-04	AG 99	1,86E-03	2,05E-03
NB 85	9,54E-03	1,00E-02	CD 99	1,34E-03	1,41E-03
MO 85	4,86E-03	4,86E-03	CD100	1,71E-03	1,86E-03
NB 86	3,30E-03	3,53E-03	CD101	1,05E-03	1,16E-03

Table 2 Results comparison of nuclei with resulting mass fraction abundance above 10^{-4}

The abundances at the end of the simulation are shown in Table 2. All orders of magnitude are the same for both the sequential and parallel codes, with a maximum variation of ~9% for ^{101}CD and an overall average variation of 5%. Note that one of the reasons for the small variations in final abundances shown in the above table is the different simulation times at which they are obtained, since the sequential and parallel versions arrive at different simulation end times when the computation completes. Whilst the parallel application final simulation time is 95.69 sec., the sequential application final simulation time is 94.94 sec. Difference is not significant, since this time difference would only affect short-lived isotopes, whereas the computation concludes at the tail of the burst (T~0.7 GK) where short lived isotopes have mostly already decayed. This accounts for a fraction of the difference in the final abundance. The rest of the small differences are directly attributable to the different methods used in the solution of the system of equations, and to numerical round-off errors.

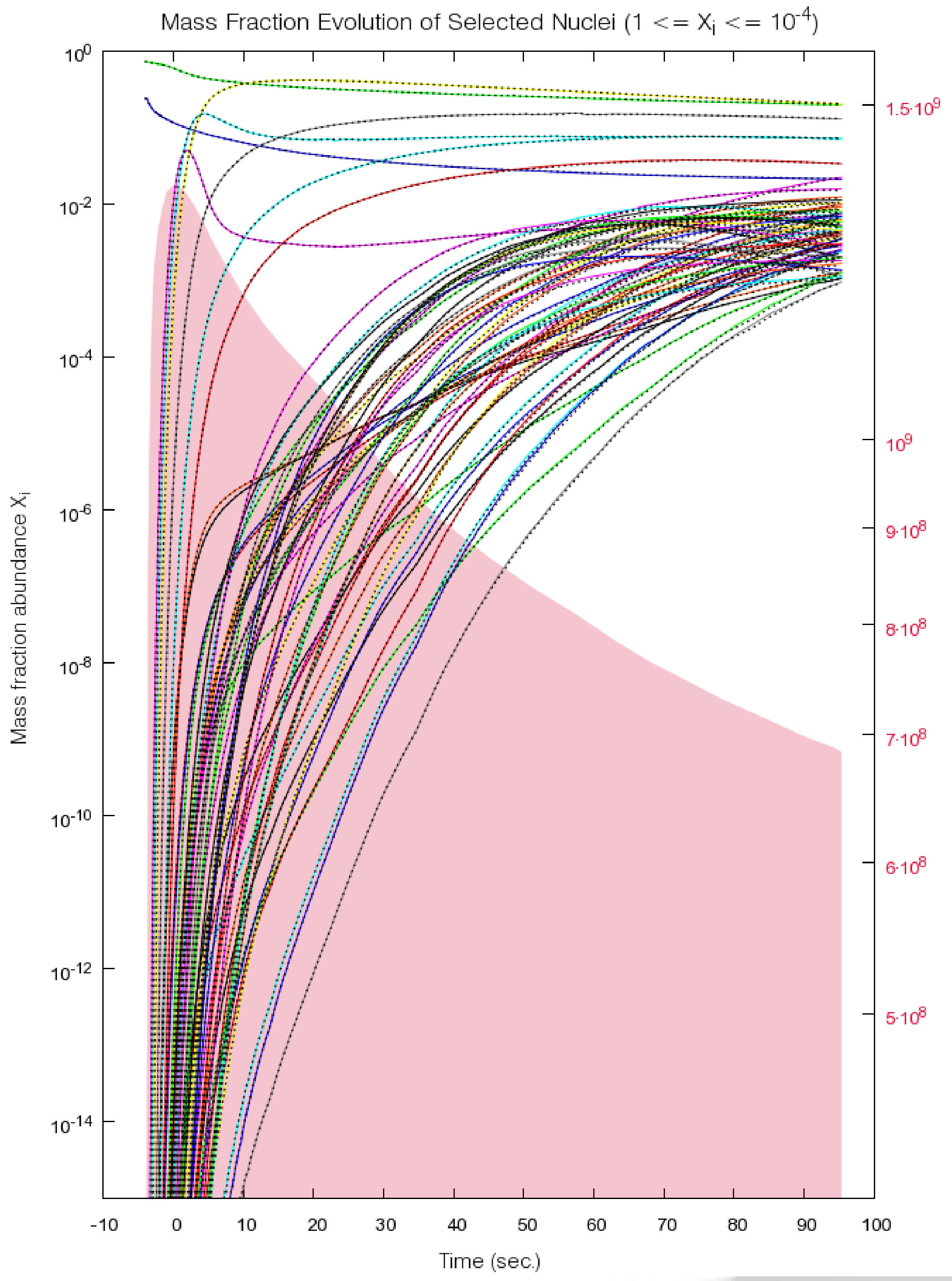
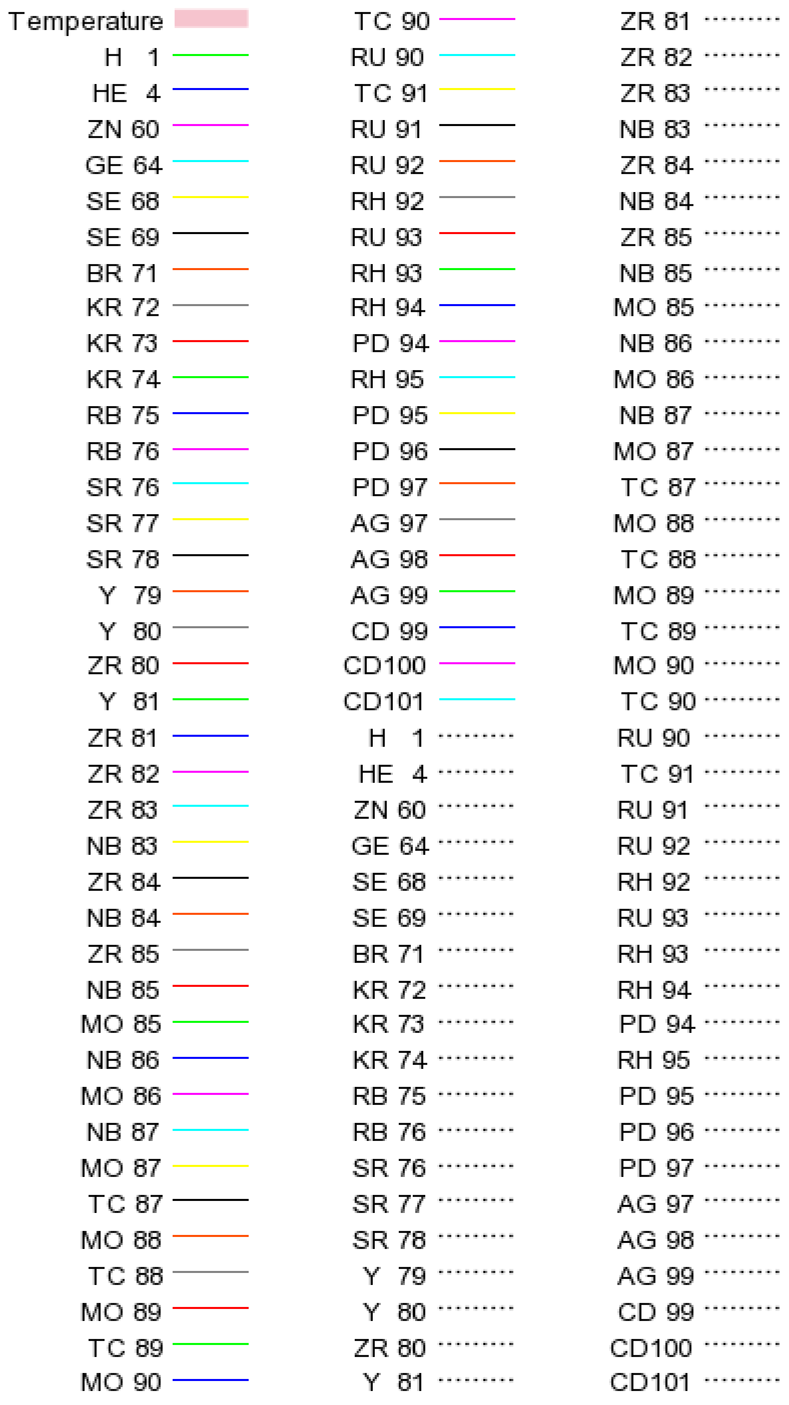


Fig. 28 Results comparison of mass fraction evolution of selected nuclei ($1 \leq X_i \leq 10^{-4}$)

The comparison of the time evolution for mass fraction of selected nuclei ($1 \leq Xi \leq 10^{-4}$) is illustrated in Fig. 28. Results arising from the parallel application are drawn using solid coloured lines (see legend in Fig. 29 below), whereas results from the sequential, 1-node application are pictured as dark dashed lines. It is evident from this results that both applications yield the same time evolution of the abundances along the time-span of the simulation. Even though there are small variations when one zooms into the graphic, these are much less pronounced for the most abundant nuclei (i.e. ^{1}H, ^{68}SE, and ^{72}KR).



Temperature	▬	TC 90	——	ZR 81	┈┈┈
H 1	——	RU 90	——	ZR 82	┈┈┈
HE 4	——	TC 91	——	ZR 83	┈┈┈
ZN 60	——	RU 91	——	NB 83	┈┈┈
GE 64	——	RU 92	——	ZR 84	┈┈┈
SE 68	——	RH 92	——	NB 84	┈┈┈
SE 69	——	RU 93	——	ZR 85	┈┈┈
BR 71	——	RH 93	——	NB 85	┈┈┈
KR 72	——	RH 94	——	MO 85	┈┈┈
KR 73	——	PD 94	——	NB 86	┈┈┈
KR 74	——	RH 95	——	MO 86	┈┈┈
RB 75	——	PD 95	——	NB 87	┈┈┈
RB 76	——	PD 96	——	MO 87	┈┈┈
SR 76	——	PD 97	——	TC 87	┈┈┈
SR 77	——	AG 97	——	MO 88	┈┈┈
SR 78	——	AG 98	——	TC 88	┈┈┈
Y 79	——	AG 99	——	MO 89	┈┈┈
Y 80	——	CD 99	——	TC 89	┈┈┈
ZR 80	——	CD100	——	MO 90	┈┈┈
Y 81	——	CD101	——	TC 90	┈┈┈
ZR 81	——	H 1	┈┈┈	RU 90	┈┈┈
ZR 82	——	HE 4	┈┈┈	TC 91	┈┈┈
ZR 83	——	ZN 60	┈┈┈	RU 91	┈┈┈
NB 83	——	GE 64	┈┈┈	RU 92	┈┈┈
ZR 84	——	SE 68	┈┈┈	RH 92	┈┈┈
NB 84	——	SE 69	┈┈┈	RU 93	┈┈┈
ZR 85	——	BR 71	┈┈┈	RH 93	┈┈┈
NB 85	——	KR 72	┈┈┈	RH 94	┈┈┈
MO 85	——	KR 73	┈┈┈	PD 94	┈┈┈
NB 86	——	KR 74	┈┈┈	RH 95	┈┈┈
MO 86	——	RB 75	┈┈┈	PD 95	┈┈┈
NB 87	——	RB 76	┈┈┈	PD 96	┈┈┈
MO 87	——	SR 76	┈┈┈	PD 97	┈┈┈
TC 87	——	SR 77	┈┈┈	AG 97	┈┈┈
MO 88	——	SR 78	┈┈┈	AG 98	┈┈┈
TC 88	——	Y 79	┈┈┈	AG 99	┈┈┈
MO 89	——	Y 80	┈┈┈	CD 99	┈┈┈
TC 89	——	ZR 80	┈┈┈	CD100	┈┈┈
MO 90	——	Y 81	┈┈┈	CD101	┈┈┈

Fig. 29 Legend of selected nuclei ($1 \leq X_i \leq 10^{-4}$)

The last element of validation is the energy released in the simulated X-ray burst. Energy contribution from all nuclear reactions taking place during the burst, are summed together to produce the final energy released. A typical curve is obtained with a very rapid increase in energy generation at the beginning of the burst, and a gradual decay as the burst proceeds; this is shown in Fig. 30 below.

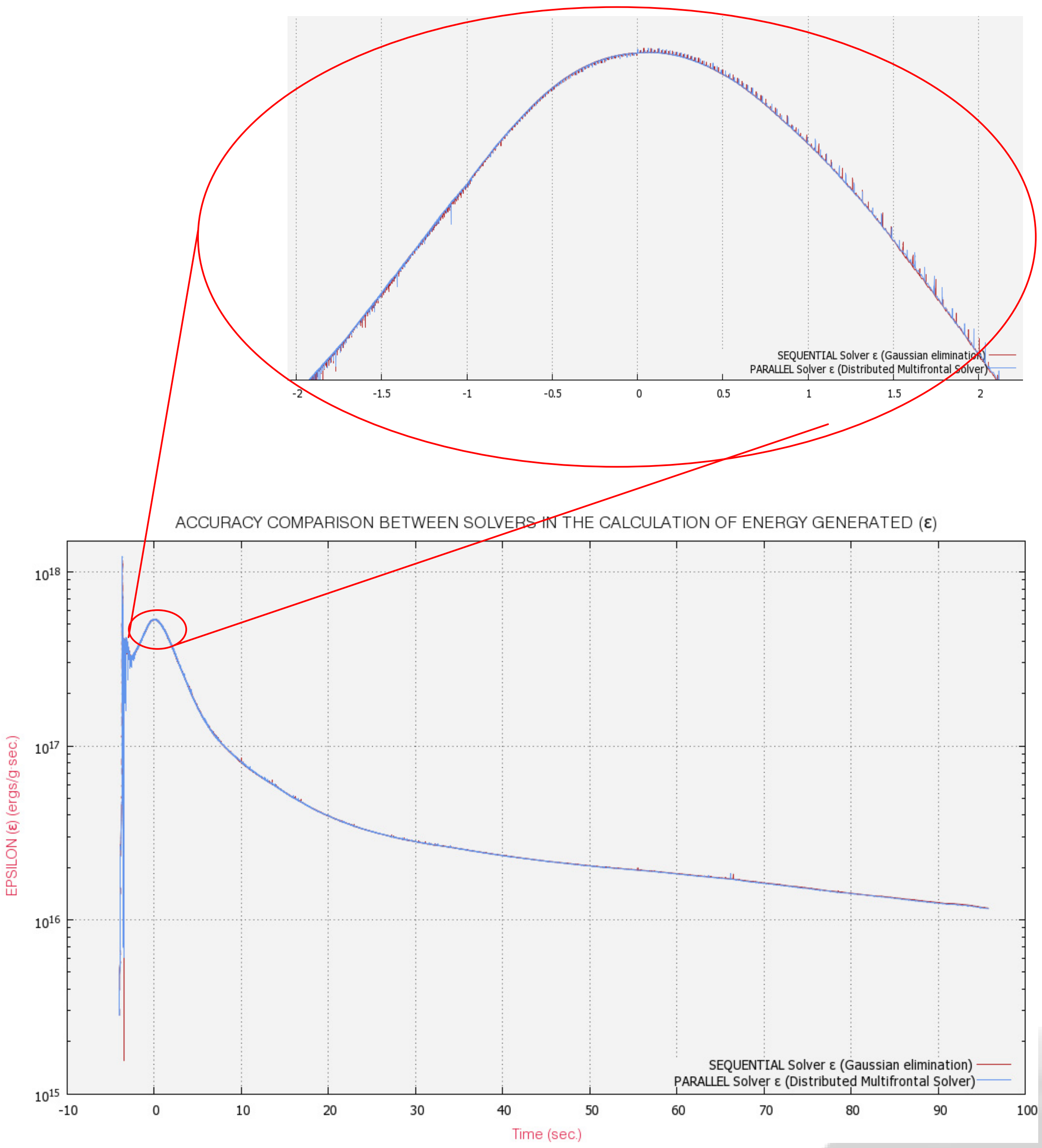


Fig. 30 Comparison of energies released in both sequential and parallel solvers

Note that the results obtained for the sequential and parallel versions of the code are almost undistinguishable, following precisely the same time evolution. It is only when we zoom in to see the detail, that some pikes and variations can be seen. These small differences are directly attributable to numerical round-off errors between the two different approaches.

3.5. Results and Discussion

The performance of the parallel execution has to be compared with the performance of the sequential application, in which the system of equations is solved by means of an iterative technique. The sequential, baseline program provides a built-in mechanism for validating the results of the parallel program, as well as a basis for calculating improvements in the performance (that is, how much faster is the parallel version with respect to the sequential version).

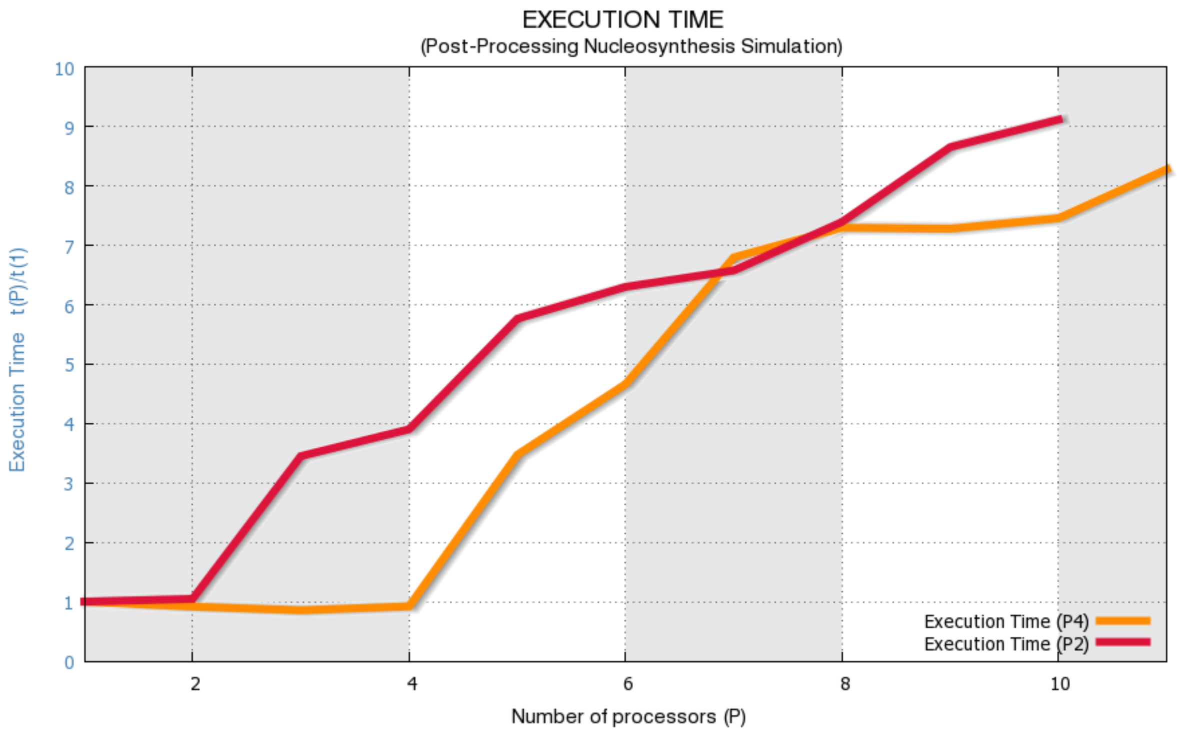


Fig. 31 Performance results: Total execution time

Being smack in the middle of the area of a loosely synchronous application (see section 2.3.3), parallelising a post-processing nucleosynthesis code poses a serious risk of obtaining worst performances than the sequential version if the ratio between computation and communication time is not properly maximised. Unfortunately, in our case the risk has materialised, and the parallel application actually takes frustratingly longer to complete than its 1-node counterpart. This is depicted in Fig. 31, showing the total execution time for an increasing number of nodes used in the simulation. The reference value (sequential version) can be found at $P = 1$ and the total execution time is depicted as the ratio between the parallel and sequential execution times ($t(P)/t(1)$). Two different executions are provided, P2 and P4, to designate an execution where the first two or four nodes, respectively, are physically located on the same machine (that is, a multiprocessor machine). Execution P2 has been obtained using only dual core workstations, whereas execution P4 has been run with one quad-core workstation for the first four nodes and the rest of the processors being hosted on dual core workstations.

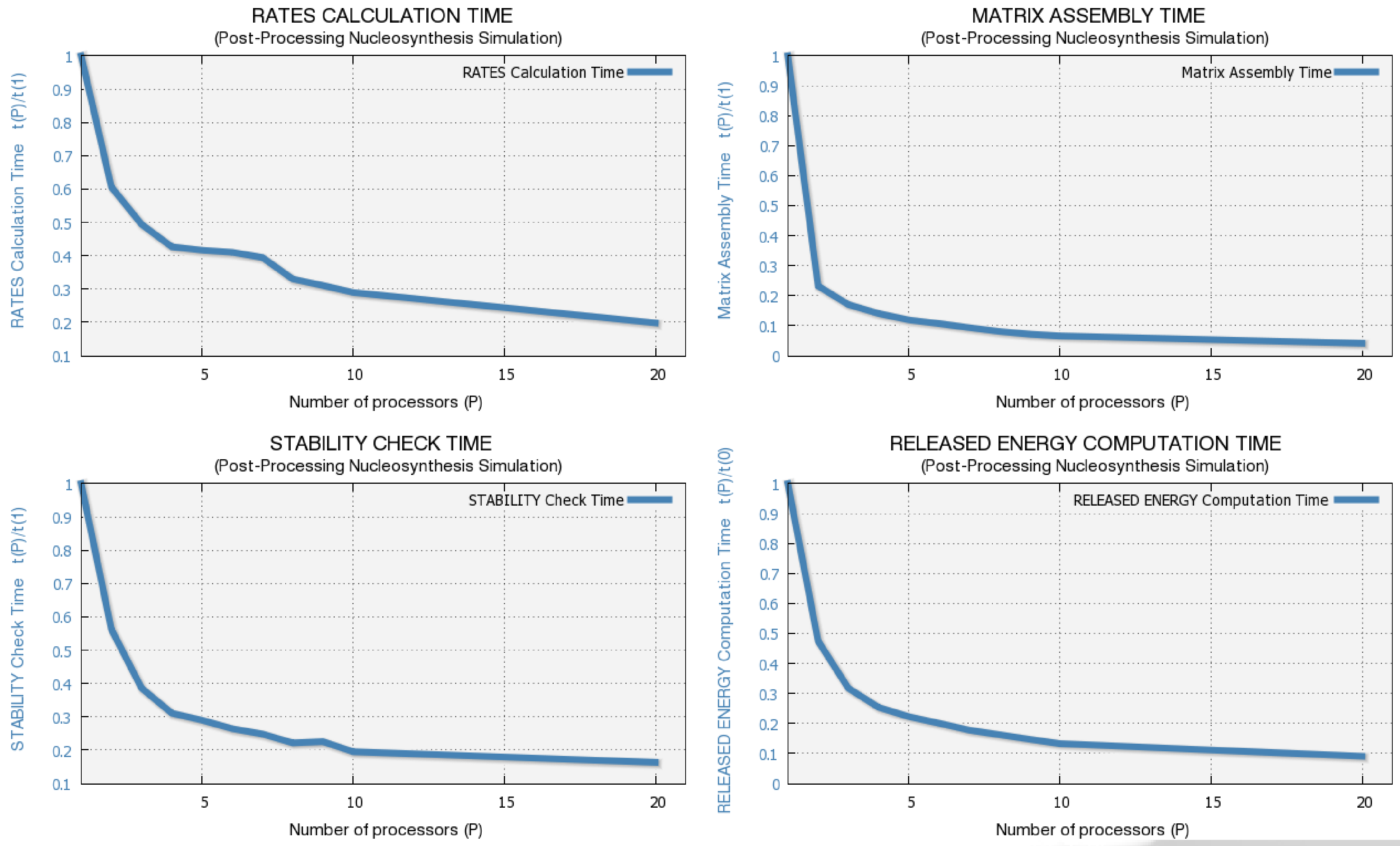


Fig. 32 Performance results: Partial execution times

It is clear that the execution time increases significantly when nodes physically separated participate in the simulation (that is, on different workstations of the cluster), whereas when the parallel application is run using nodes of the same machine, the execution time is kept at bay with respect to the sequential execution time, and even small speed-ups are obtained when using a quad core machine, for two, three and four nodes.

We set off to determine where is the parallel execution taking most of the time, and which part of the simulation is to be blamed for the significantly longer execution times. Fig. 32 shows the partial execution times of the *rates calculation* (upper left), *matrix assembly* (upper right), *stability check* (bottom left), and *energy released* (bottom right) stages of the simulation (see section 3.3). It is clear from these curves that the parallelisation strategy for these parts is excellent and that significant speed-ups are obtained in all cases when the processing stage is run in parallel. Execution times are shown as the ratio between the parallel and sequential execution times $(t(P)/t(1))$. For instance, the matrix assembly runs almost five times faster using five nodes in parallel than just one node taking care of all the matrix assembly work, and completes almost seven times faster when using ten nodes in the assembly of the matrix. This is very close to a nearly-ideal speed-up (see section 2.4).

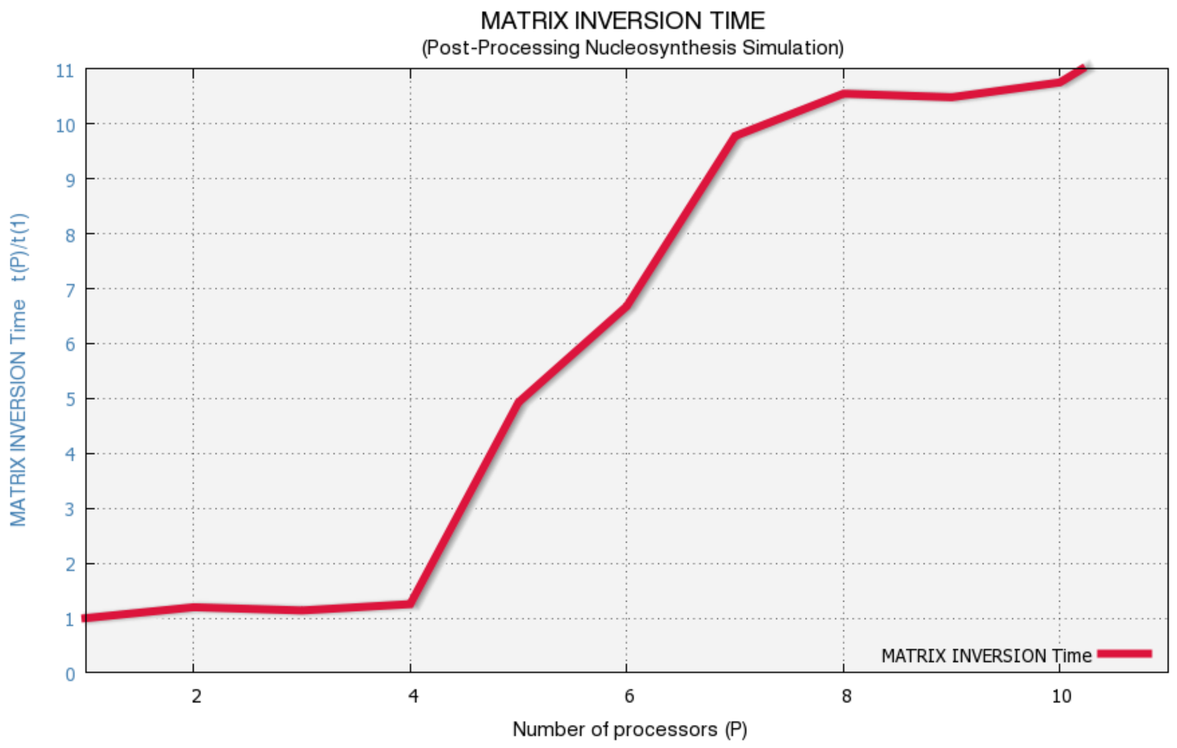


Fig. 33 Performance results: Matrix inversion time

The stability check and energy released computation time also yield valuable increases in performance (although smaller than the matrix assembly case). Both run consistently faster in the parallel version than in the sequential application. Accordingly to the discussion held in section 3.3.2 the interpolation of reaction rates gives the smallest benefits in terms of an increase of performance (due to overlapping reaction rates being calculated by several nodes at the same time). However, even in this case, the rise in performance execution is evident, for instance, with a speed-up of 2.5 using 5 nodes in the parallel execution. Consequently, these stages are not to be blamed in the performance lost when the total execution time is taken into account.

Fig. 33 shows the performance results for the matrix inversion time (see section 3.3.4). In this case the behaviour bears little in common with the other computation stages; the solution of the system of equations takes consistently longer for the parallel application for any number of nodes used in the computation. Note that for the matrix inversion, we do not even get the small improvements of using nodes physically located on the same machine. Even though the execution time is more or less controlled up to four nodes (the simulation has been obtained using a quad-core machine), the performance plummets dramatically for a higher number of processors.

So the main responsible for loosing so much performance seems quite clearly the solution phase of the system of equations. However, since the matrix $\mathbf{A}$ is perfectly distributed amongst processors, with no overlapping entries (see section 3.3.3); it is conceivable to think that the total time strictly devoted to computation in the parallel solution should not be much longer than the time used to solve the system of equations in the sequential version. There might be an overhead in the case of the parallel application, due to the multifrontal decomposition and task assignment to the several processors, but by no means this increase in the computation time accounts for all the loss in performance.

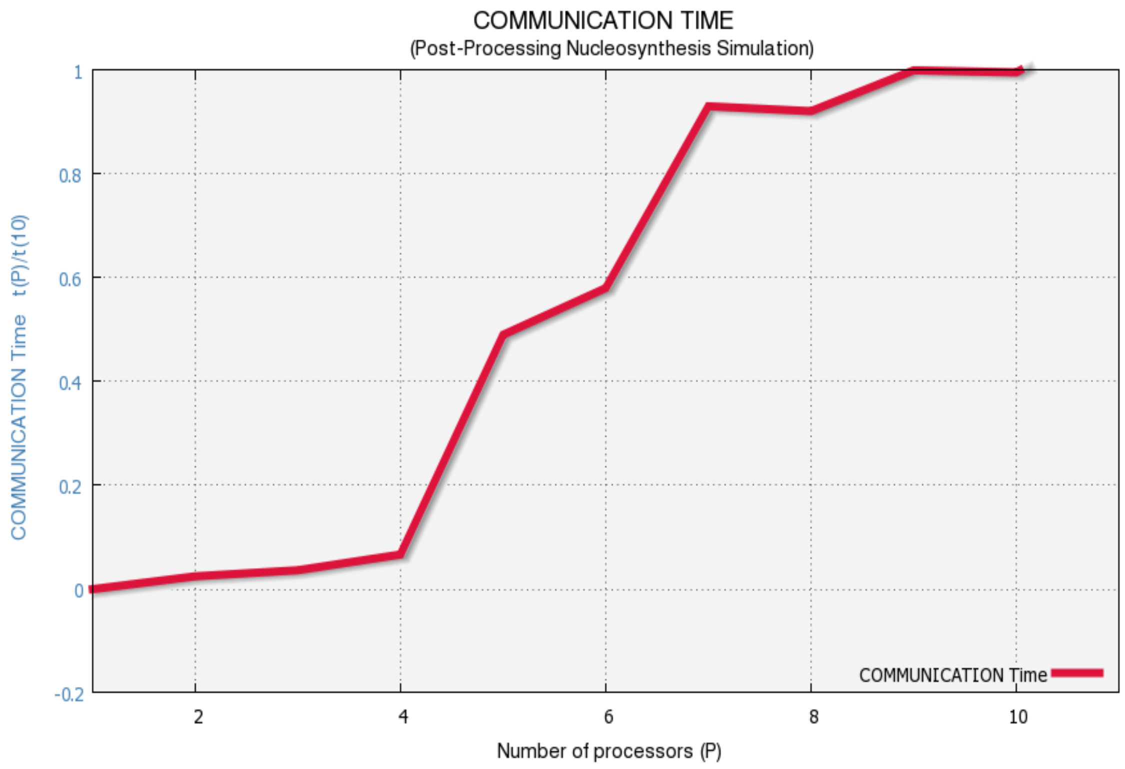


Fig. 34 Performance results: Communication time

There is still one more component to be investigated, namely the overall communication time. In order to get an estimation of the evolution of the communication time for the parallel application, the time spent on communication points 2, 3 and 4 (see Fig. 18) has been measured and it is depicted in Fig. 34. It can be clearly appreciated the same pattern that underlies the matrix inversion and total execution times; the communication time increases slightly from one to four nodes (using a quad-core machine for the first four nodes), but soars rapidly whenever physically separated nodes are incorporated into the parallel execution. This result is consistent with the multiprocessor architecture described in section 2.3.1, where it was claimed that communication times had to be shorter for nodes sharing resources (common memory, communication buses, etc) than for those that were not physically located in the same machine, due to network latencies, and the overhead posed by the communication protocol. Also, implementations of the message passing interface (MPI) have been generally designed to detect when the nodes are physically located on the same machine, therefore optimising the buffering, synchronisation, and message passing mechanism for those nodes (MPI Forum 2009, and MPICH2 2011).

Note that in Fig. 34 it is not included the communication time spent within the matrix inversion stage. This is due to the fact that MUMPS does not provide a mechanism to inform the user of the time spent on message passing. Be that as it may, and in light of the behaviour of the performance of the solution of the system of equations (Fig. 33), it will most possibly follow the same pattern as the one shown in Fig. 34. We can hence safely arrive at the verdict that the high communication costs, together with a relatively limited computation time, are the main responsible for the loss in performance.

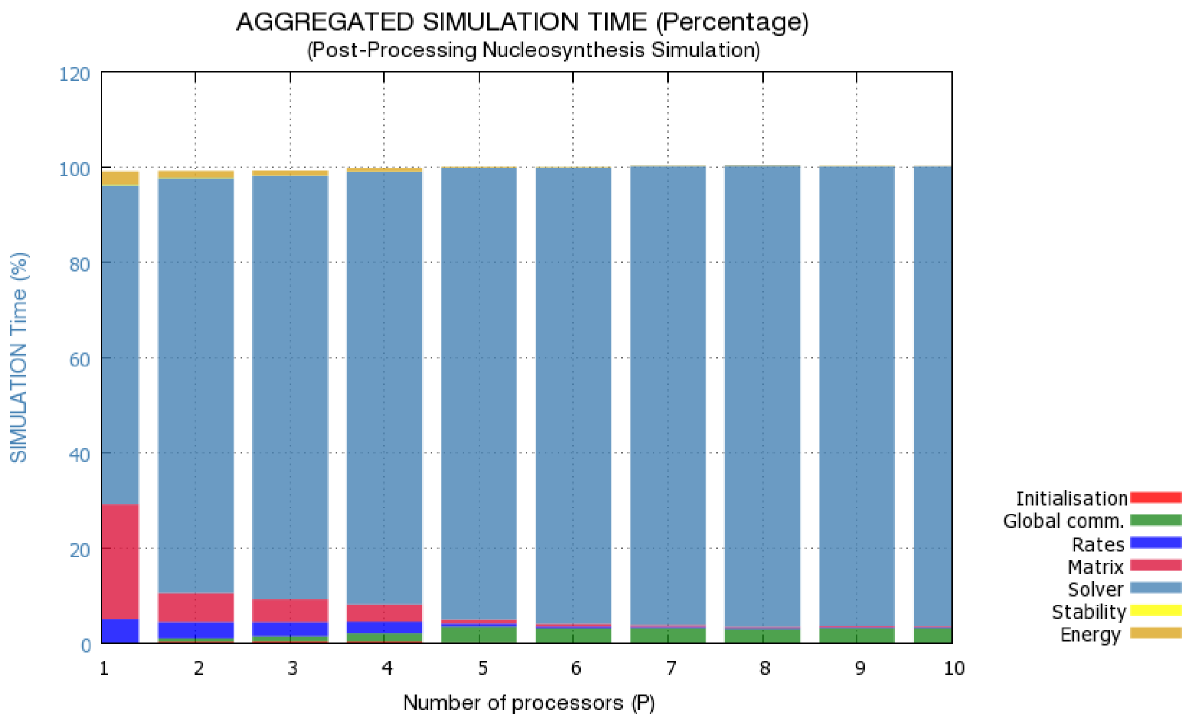


Fig. 35 Performance results: Aggregated simulation time (percentage)

It is still left the question of why the gains in performance in the stages of rates calculation, matrix assembly, stability check, and released energy do not make up for the increase in communication times. Fig. 35 shows the percentage of the total simulation time devoted to initialisation, global communications (not including MUMPS internal communications during the solution of the system of equations phase), rates calculation, matrix assembly, stability check, energy generation, and matrix inversion (solver). The sequential execution spends most of the time inverting the matrix (82%) and building (15%) the system of equations. Energy generation accounts for just 1% of the total computation time. The relative time spent on the interpolation of reaction rates is just a 0.44% of the total execution time, whereas only 0.06% of the time is spent on stability check. With an increasing number of nodes participating in the simulation, the time spent on global communications and in the solution of the system of equations gradually tends to account for nearly all computation time, so the time spend in the other processing stages becomes virtually negligible with respect to the former. This is the reason why the improvements in performance in matrix assembly, interpolation of reaction rates, stability check and energy generation, are close to nothing when compared with the cost of global communications.

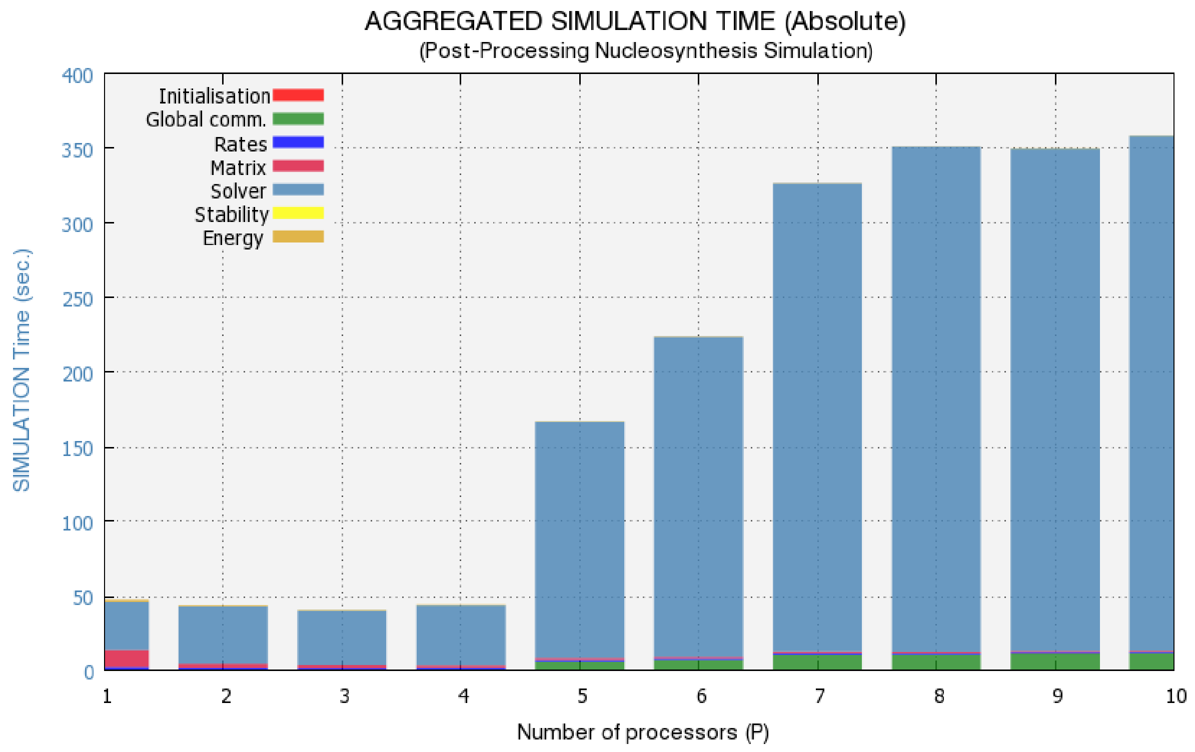


Fig. 36 Performance results: Aggregated simulation time (absolute)

To put this in context, Fig. 36 shows the absolute aggregated simulation time[8], where it is seen that the time spent on stages other than global communications and solving the system of equations that arises from the linearisation of the set of differential equations describing the time evolution of the network abundances is negligible.

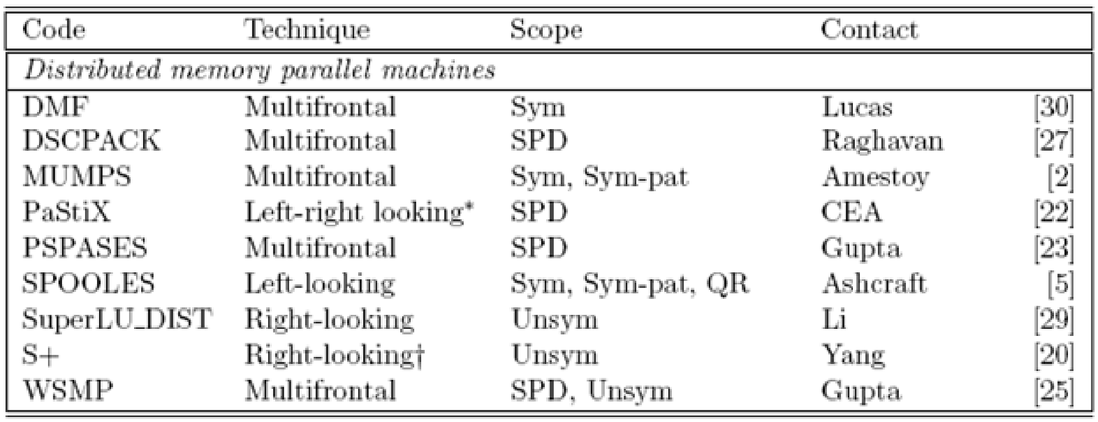
3.6. Discussion on the Chosen Solver

Results of previous sections reveal that the solution phase of the system of equations takes up nearly all the execution time in the parallel version of the code. Having such a loss in performance, it is compulsory to analyse whether the selection of MUMPS as a solver has been appropriate. In other words, it must be analysed whether MUMPS is to be blamed for the loss in the performance of the parallel application.

MUMPS (MUltifrontal Massively Parallel sparse direct Solver) is a software application for the solution of large sparse systems of linear algebraic equations Ax = b on distributed memory parallel computers. It was initially funded by LTR (Long Term Research) European project PARASOL (1996-1999), and it is currently used widely by industries (Boeing, EADS, EDF,

[8] For comparison purposes, a simulation with only 5000 models has been considered.

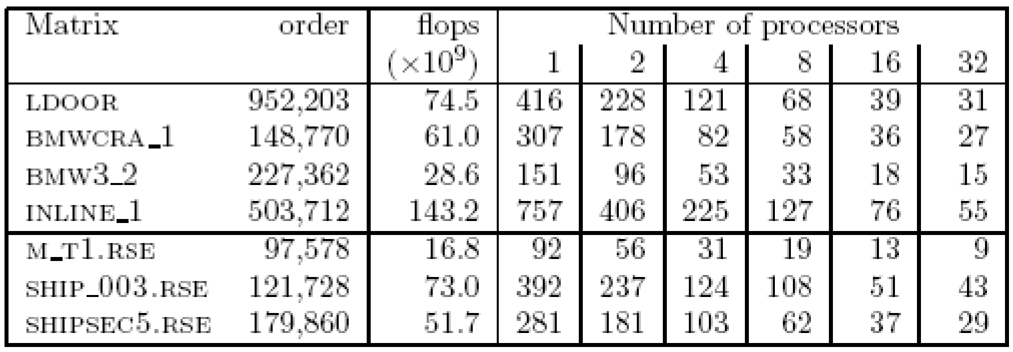
Petroleum industries, Buttari *et al.* 2010), and numerical simulations of fusion plasma (Åström 2009). It has been also integrated within commercial and academic packages, and on top of that MUMPS represents one of the scarce professional and supported public domain implementations of the multifrontal method.



Code	Technique	Scope	Contact	
Distributed memory parallel machines				
DMF	Multifrontal	Sym	Lucas	[30]
DSCPACK	Multifrontal	SPD	Raghavan	[27]
MUMPS	Multifrontal	Sym, Sym-pat	Amestoy	[2]
PaStiX	Left-right looking*	SPD	CEA	[22]
PSPASES	Multifrontal	SPD	Gupta	[23]
SPOOLES	Left-looking	Sym, Sym-pat, QR	Ashcraft	[5]
SuperLU_DIST	Right-looking	Unsym	Li	[29]
S+	Right-looking†	Unsym	Yang	[20]
WSMP	Multifrontal	SPD, Unsym	Gupta	[25]

Fig. 37 Alternatives for direct sparse solvers

There are other alternatives for direct sparse solvers, as shown in Fig. 37 (Li 2010). However, some of them only work with symmetric positive definite matrices (DSCPACK, PaStiX, and PSPASES); others are limited to working only with symmetric matrices (DMF, and SPOOLES). SuperLU_DIST and S+ have only a slightly worst performance than MUMPS, but they are less widely used by industry and researchers. Finally, WSMP (Gupta 2002) has reportedly a similar performance to that of MUMPS (Gupta *et al.* 2001), but it is a commercial IBM solution (the fee for a perpetual license is $16K for 512 - or unlimited - cores), and the extra cost does not pay off for the very little increase in performance.

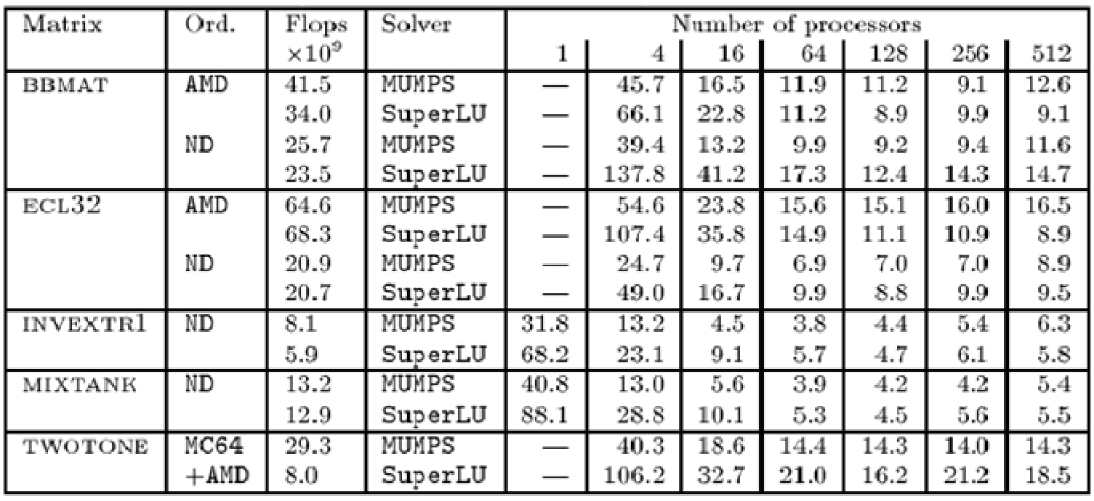


Matrix	order	flops	Number of processors					
		$(\times 10^9)$	1	2	4	8	16	32
LDOOR	952,203	74.5	416	228	121	68	39	31
BMWCRA_1	148,770	61.0	307	178	82	58	36	27
BMW3_2	227,362	28.6	151	96	53	33	18	15
INLINE_1	503,712	143.2	757	406	225	127	76	55
M_T1.RSE	97,578	16.8	92	56	31	19	13	9
SHIP_003.RSE	121,728	73.0	392	237	124	108	51	43
SHIPSEC5.RSE	179,860	51.7	281	181	103	62	37	29

Fig. 38 MUMPS solver performance for large matrices

The MUMPS solver performance for large matrices is excellent (Fig. 38, Amestoy *et al.* 2000). For matrices of order $\geq$ 100k, very good speed-ups are accomplished using MUMPS (e.g. between 2.8 up to 3.7 with 4 processors). And even higher speed-ups are attained with higher number of processors (e.g. between 7.1 up to 10.6 with 16 processors). Note that speed-ups increase with matrix size, in a similar way as big images get higher compression rates than smaller images.

In the case of medium sized matrices, the solver performance is still very good (Fig. 39, Amestoy *et al.* 2001b). For matrices of the order between 10k and 100k, good speed-ups are accomplished using MUMPS (e.g. between 2.4 up to 3.1 with 4 processors, depending on the matrix size). Again, higher speed-ups are attained with higher number of processors (e.g. between 7.2 up to 8.4 with 16 processors).



Matrix	Ord.	Flops $\times 10^9$	Solver	Number of processors						
				1	4	16	64	128	256	512
BBMAT	AMD	41.5	MUMPS	—	45.7	16.5	11.9	11.2	9.1	12.6
		34.0	SuperLU	—	66.1	22.8	11.2	8.9	9.9	9.1
	ND	25.7	MUMPS	—	39.4	13.2	9.9	9.2	9.4	11.6
		23.5	SuperLU	—	137.8	41.2	17.3	12.4	14.3	14.7
ECL32	AMD	64.6	MUMPS	—	54.6	23.8	15.6	15.1	16.0	16.5
		68.3	SuperLU	—	107.4	35.8	14.9	11.1	10.9	8.9
	ND	20.9	MUMPS	—	24.7	9.7	6.9	7.0	7.0	8.9
		20.7	SuperLU	—	49.0	16.7	9.9	8.8	9.9	9.5
INVEXTR1	ND	8.1	MUMPS	31.8	13.2	4.5	3.8	4.4	5.4	6.3
		5.9	SuperLU	68.2	23.1	9.1	5.7	4.7	6.1	5.8
MIXTANK	ND	13.2	MUMPS	40.8	13.0	5.6	3.9	4.2	4.2	5.4
		12.9	SuperLU	88.1	28.8	10.1	5.3	4.5	5.6	5.5
TWOTONE	MC64	29.3	MUMPS	—	40.3	18.6	14.4	14.3	14.0	14.3
	+AMD	8.0	SuperLU	—	106.2	32.7	21.0	16.2	21.2	18.5

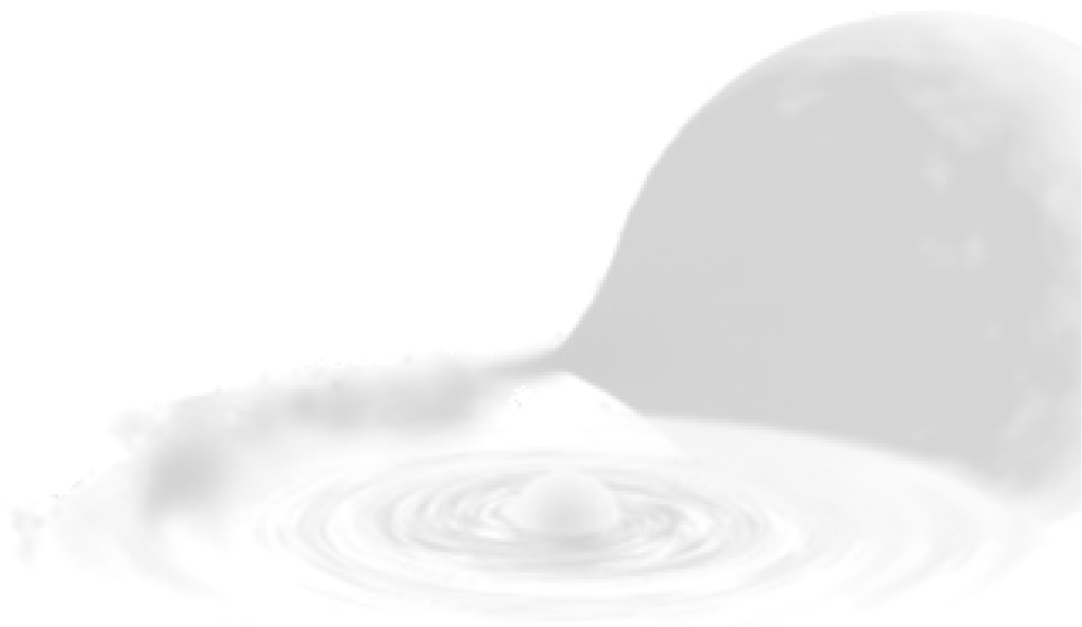
Fig. 39 MUMPS solver performance for medium sized matrices

Not much data can be found in the literature with regards to the performance of the parallel solvers for matrices of order ≤10k. This is due to the fact that as the problem dimension shrinks, the distributed computation time is also reduced; whilst communication time diminishes much less noticeably. This provokes that the accomplished speed-ups are greatly affected. For instance, Fox (2007) reports speed-ups of 1 (i.e. no speed-up at all) with 4 processors, and a speed-up of 1.8 for 16 processors, in solving a system with 5.535 elements using the MUMPS solver.

In light of these results, it seems clear that the performance of the MUMPS solver is not to be put into question, but rather, the order of the nucleosynthesis matrix, which is too small to maximise the ratio between computation time and communication time. A long story short; the sequential application takes much shorter time in solving the system of equations, than the time that the parallel application spends in communications. The linearised system of equations of the network abundances derivatives is a small, sparse matrix whose order is limited by the number of isotopes of the nucleosynthesis network (NIS = 606 in this Work). Even if we were to increase the number of isotopes of the nucleosynthesis network, in an attempt to maximise the computation to communication ratio, maybe we could increase the order of the matrix $\mathbf{A}$ up to roughly a thousand nuclides. This is still too small a number so that the problem size can be increased to a point where speed-ups are accomplished in the parallel solution of the system of equations. The main conclusion to be drawn here is that the parallelisation of a post-processing nucleosynthesis code is therefore not worth the effort.

As a next step, we will parallelise the fully coupled hydrodynamic code (which falls into the category of a fully synchronous application), where better opportunities for parallelisation exist. This will be dealt with in the following chapter.

This Page Intentionally Left Blank



CHAPTER 4
PARALLELISATION OF A HYDRODYNAMIC CODE

"There are now three types of scientists: experimental, theoretical, and computational."

Silvan S. Schweber

Quoted by Victor F. Weisskopf, 'One Hundred Years of the Physical Review', in H. Henry Stroke, Physical Review: The First Hundred Years: a Selection of Seminal Papers and Commentaries, Vol. 1, 13.

"Conversion of any code to parallel takes a few weeks, perhaps longer."

Ed Barsis

Quote collected by Steve Plimpton in the Massively Parallel Computing Research Laboratory at Sandia National Laboratories.

4.1. Introduction

Hydrodynamic calculations of type I X-ray bursts and their associated nucleosynthesis have been extensively addressed by different groups (see for instance early models by Woosley & Taam 1976, Maraschi & Cavaliere 1977, and Joss 1977), which shows the great scientific interest that is posed onto determining the processes that trigger the thermonuclear runaways as well as in the determination of the final composition of the neutron star surface right after the explosion. In addition, several thermal, radiative, electrical, and mechanical properties of the neutron star depend critically on the specific chemical abundance pattern of its outer layers.

In order for this simulation to be as realistic as possible, it is necessary to make use of a complete hydrodynamic code, coupled with a fully updated nuclear reaction network, so that the model is capable of self adjusting both the temperature and density of the stellar envelope according to the nuclear reaction processes that take place in the surface of the neutron star. The scale of this fully coupled models, usually make the simulation computationally prohibitive for large reaction networks, and therefore the scientific community often has to resort to using a reduced nuclear reaction network truncated around Ni (Woosley & Weaver 1984; Taam *et al.* 1993; Taam *et al.* 1996 –all using a 19-isotope network), Kr (Hanawa *et al.* 1983 –274-isotope network; Koike *et al.* 1999–463 nuclides), Cd (Wallace & Woosley 1984 –16-isotope network), or Y (Wallace & Woosley 1981 –250-isotope network). On the other hand, Schatz *et al.* (1999, 2001a) have carried out very detailed nucleosynthesis calculations with a network containing more than 600 isotopes (up to Xe, in Schatz *et al.* 2001a), but using a one-zone approach (Woosley *et al* 2004, José *et al.* 2010, Fisker *et al.* 2006).

One of the main goals of this Master Thesis is to successfully parallelise the spherically symmetric, Lagrangian, hydrodynamic code SHIVA (José 1996; José & Hernanz 1998), in pursuit of significant speed-ups that allow for detailed hydrodynamic simulations with extended nuclear reaction networks in affordable times. It was discussed in section 2.3.2 that this problem architecture represents the so called fully synchronous parallelism, indicating that (at least in principle) each computation is performed synchronously (or simultaneously) to all data. The main point here is that all future calculations of decisions hinge on the results of the earlier, preceding data calculations. Parallelisation can be achieved by having each node actually cycling through a subset of the neutron star envelope shells (i.e. a number of contiguous shells). If this group of shells, assigned to each processor, is not homogeneous, the workload may vary across different nodes. Fig. 12 showed that this type of problem architecture is more suitable for parallelisation, hence better results are expected than those obtained with the parallelisation of the post-processing nucleosynthesis code (see 3.5 - Results).

4.2. Hydrodynamic Simulation Code: SHIVA

4.2.1. Application Description

The hydrodynamic simulation code to be parallelised in this Thesis is a modified version of SHIVA, a one-dimensional (spherically symmetric), hydrodynamic code, in Lagrangian formulation, built originally to model classical nova outbursts (José 1996; José & Hernanz 1998). A flow chart describing the basic structure of the SHIVA code is outlined in Fig. 41. The code uses a co-moving coordinate system, where time derivatives of any variable are calculated with respect to a grid attached to the fluid, as described in Kutter & Sparks (1972). This formulation avoids the spurious generation of numerical diffusion, which causes many problems in the attempt to model burning fronts.

Despite convective mixing has certainly a multi-dimensional nature, most of the main observational features that characterize type I X-ray bursts (XRBs) can be reproduced by spherically symmetric models. From a hydrodynamical viewpoint, nova outbursts and XRBs are similar objects: both are powered by thermonuclear explosions driven by mass accretion on the surface of a compact star (a white dwarf, in the case of a nova; a neutron star, for an XRB). Although the basic stellar structure equations governing nova explosions and XRBs are identical, the different surface gravity (much stronger in a neutron star) induces dramatic differences in the physical conditions that define such cataclysmic events.

4.2.2. Shell Structure

In the simulations, the outermost layers of the neutron star are divided into N concentric mass shells (with intershells labelled with a subscript i, ranging from 1, at the very centre -or innermost shell- of the star, to $N+1$ at the surface; see Fig. 40). This structure defines a Lagrangian grid, where the mass interior to the ith-intershell, m_i, and the star's age, t, are taken as the independent variables. The code computes the time evolution of several physical variables, such as the luminosity, L, the radius, r, the velocity, v, the temperature, T, and the density, ρ, for each shell. Following Kutter & Sparks (1972), L, r, and v are evaluated at the intershells (and are denoted by subscripts i), whereas other variables, such as T, or ρ, are shell-centred (i.e. evaluated at mass points defined by geometric averages, as $m_{i+1/2} = \sqrt{m_{i+1} \cdot m_i}$, and denoted by half-integer subscripts $i+1/2$). The time step is defined as $\Delta t^{n+1/2} = t^{n+1} - t^n$, where t^n represents the time elapsed since the beginning of the simulation.

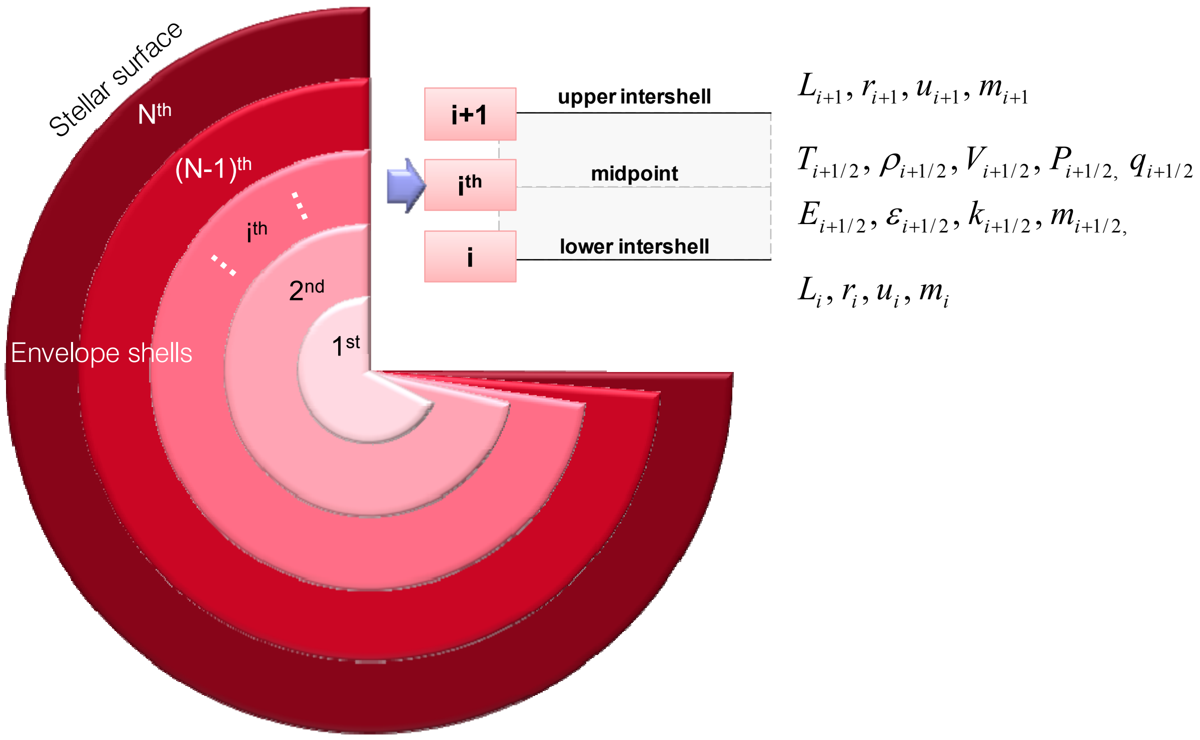

Stellar surface
N th
(N-1) th
i th
2 nd
1 st
Envelope shells
i+1
i th
i
upper intershell
midpoint
lower intershell
$L_{i+1}, r_{i+1}, u_{i+1}, m_{i+1}$
$T_{i+1/2}, \rho_{i+1/2}, V_{i+1/2}, P_{i+1/2}, q_{i+1/2}$
$E_{i+1/2}, \varepsilon_{i+1/2}, k_{i+1/2}, m_{i+1/2},$
L_i, r_i, u_i, m_i

Fig. 40 Shell structure and assignment of variables at grid points.

4.2.3. Computation Flow

From an initial converged model the software computes the EOS of matter, opacities and artificial viscosity ($P^t_{i+1/2}$, $q^t_{i+1/2}$, $E^t_{i+1/2}$, $k^t_{i+1/2}$, $\varepsilon^t_{i+1/2}$). Using these variables, the linearised system of equations for the physical values is constructed using the values at the current time t (L_i, r_i, $\rho_{i+1/2}$, u_i, $T_{i+1/2}$)[9]. The solution to the system of $5N$ equations is obtained by means of the Henyey's Method (Henyey *et al.* 1964), an iterative implicit technique because the structure equations have to be solved in parallel with the energy transport equations. The solution to the system of equations yields the new physical values at time $t+\Delta t$ ($T^{t+\Delta t}_{i+1/2}, \rho^{t+\Delta t}_{i+1/2}$). In particular, the values of temperature and density for each of the shells of the envelope allow for the computation of the nuclear abundances evolution and energy generation. This is computed for each of the shells. The next step develops the accretion of matter, $\Delta M = \dot{\mathrm{M}} \cdot \Delta t$ establishing a new mass grid and computing a mass rezoning within the envelope preserving the total number of mass shells. Finally the new variables have to be extrapolated due to the new mass grid and accretion ($L_i'^{(t+\Delta t)}$, $r_i'^{(t+\Delta t)}$, $\rho'^{(t+\Delta t)}_{i+1/2}$, $u_i'^{(t+\Delta t)}$, $T'^{(t+\Delta t)}_{i+1/2}$, $X'^{(t+\Delta t)}_i$) and a new iteration proceeds.

Note that in the flow chart of the SHIVA code shown in Fig. 41 overleaf, there appear many other steps that have been omitted for the sake of conciseness. For a more detailed description of the computing flow of the SHIVA code, see José 1996; José & Hernanz 1998, and Moreno 2009.

[9] Values at current time 't' are indicated without super index; $L_i^t = L_i$

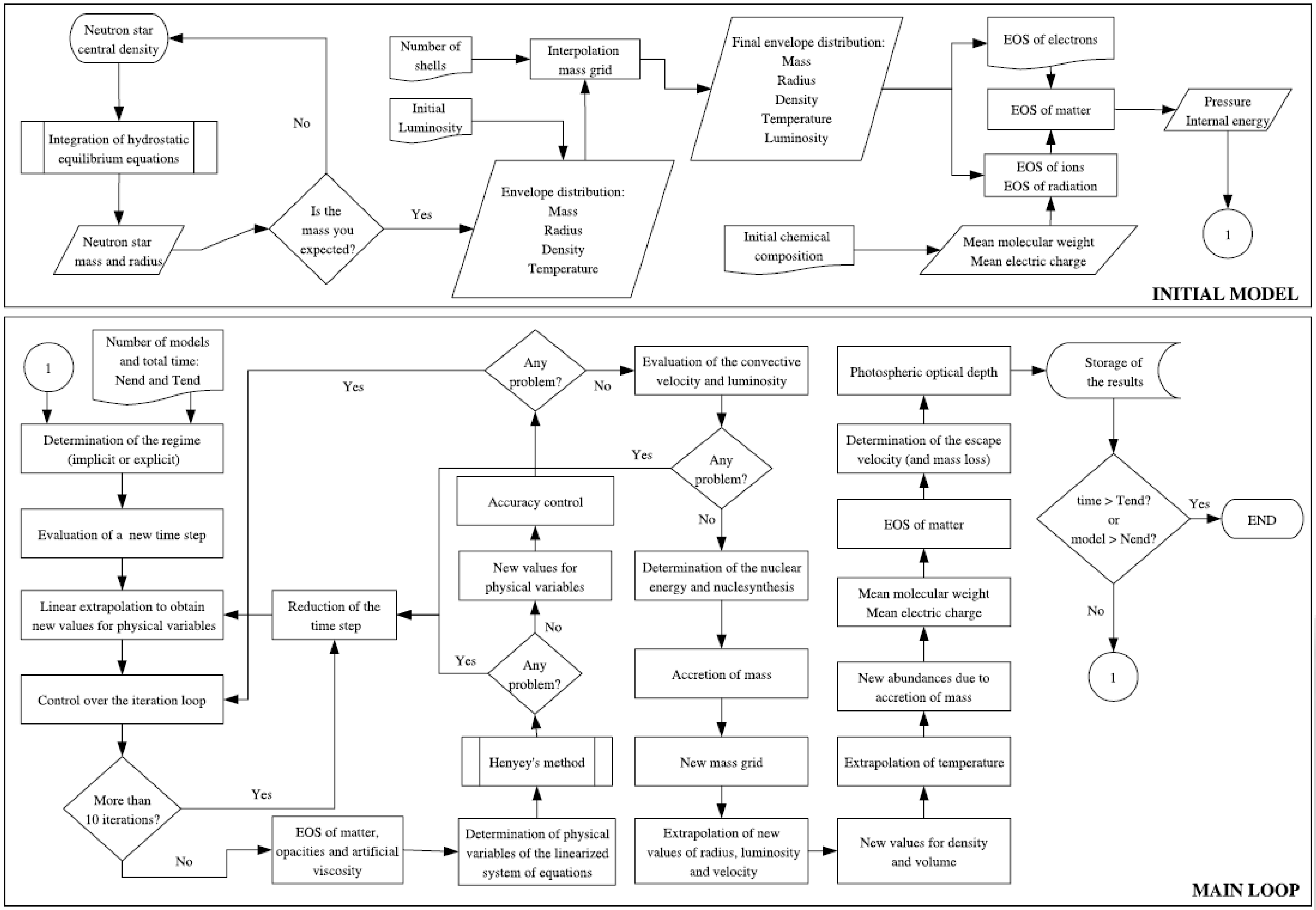

Neutron star central density
Integration of hydrostatic equilibrium equations
Neutron star mass and radius
Is the mass you expected?
No
Yes
Number of shells
Initial Luminosity
Interpolation mass grid
Envelope distribution: Mass Radius Density Temperature
Final envelope distribution: Mass Radius Density Temperature Luminosity
EOS of electrons
EOS of matter
EOS of ions EOS of radiation
Pressure Internal energy
Initial chemical composition
Mean molecular weight Mean electric charge
1
INITIAL MODEL
1
Number of models and total time: Nend and Tend
Determination of the regime (implicit or explicit)
Evaluation of a new time step
Linear extrapolation to obtain new values for physical variables
Control over the iteration loop
More than 10 iterations?
Yes
No
Reduction of the time step
EOS of matter, opacities and artificial viscosity
Determination of physical variables of the linearized system of equations
Henyey's method
Any problem?
Yes
No
New values for physical variables
Accuracy control
Any problem?
Yes
No
Evaluation of the convective velocity and luminosity
Any problem?
Yes
No
Determination of the nuclear energy and nuclesynthesis
Accretion of mass
New mass grid
Extrapolation of new values of radius, luminosity and velocity
New values for density and volume
Extrapolation of temperature
New abundances due to accretion of mass
Mean molecular weight Mean electric charge
EOS of matter
Determination of the escape velocity (and mass loss)
Photospheric optical depth
Storage of the results
time > Tend? or model > Nend?
Yes
END
No
1
MAIN LOOP

Fig. 41 Flow chart of the SHIVA code

4.3. Hydrodynamic Parallelisation

4.3.1. Parallelisation Analysis

A first analysis of the flow chart of the SHIVA code reveals two main points where parallelisation might be exploited; on the one hand the solution of the linearised system of equations for the determination of the physical variables (Henyey's method) and on the other hand the determination of the nuclear energy and nucleosynthesis, as these are computed independently for each of the shells, given a value obtained for the temperature T and density ρ for the shell. There are other stages that could be computed in parallel (e.g. construction of the system of equations, extrapolation of variables, etc.), but the experience acquired in the parallelisation of the post-processing code (Chapter 3) discourages us from tempting to parallelise routines that are not computationally intensive or that do not fall into one of the *parallelisable* problem architectures (perfect parallel, pipeline, or fully synchronous problem architectures), as the risk of not obtaining speed-ups in the execution is considerably high.

It was discussed in section 3.6 that the parallel solution of a system of linear equations only achieves acceptable performance for matrices of order $\geq$10k elements. As the problem dimension shrinks, the distributed computation time is also reduced; whilst communication time diminishes much less noticeably. This provokes that the speed-ups are greatly affected because the communication times largely exceed the distributed computation time. In our case, the order of the system of equations is $5N$, being N the number of shells used in the simulation. In our simulations we have used $N = 200$ shells which generate a system of 1000 unknowns. In light of the results from the previous chapter, this is clearly not sufficient to attempt the parallel solution of the linearised system of equations for the physical variables. Consequently, the only candidate that we will consider for parallelisation is nucleosynthesis and nuclear energy generation. This processing stage is a clear example of a perfect parallel application since, given a value of T and density ρ for each shell, and a network of nuclear reactions, the evolution of the mass fraction abundances can be calculated independently for each shell.

4.3.2. Performance Prediction

As discussed in section 2.4 it is necessary to assess the application's potential for parallelisation, and the kind of performance that is achievable for the application at hand, in order to avoid wasting resources in the parallelisation of an application that is not going to yield significant speed-ups. Performance estimates are based on timings of the baseline program, so let T_S be the total execution time of the serial application, T_{pp} the serial execution time of the potentially parallel portion of the code (in our case the total time spent in the nuclear subroutine), T_{in} and T_{out} the initialisation and output times (see Fig. 42), and $\mathrm{N_p}$ the number of processors participating in the parallel computation. Assuming perfect parallelisation of the potentially parallel portion of the code (that is, neglecting communication costs), the maximum theoretical speed-up is given by the expression:

$$\text{Speed - up} \approx \frac{\mathrm{T_S}}{\mathrm{T_{in}} + \dfrac{\mathrm{T_{PP}}}{\mathrm{N_p}} + \mathrm{T_{out}}} = \frac{\mathrm{T_S}}{\dfrac{\mathrm{T_{PP}}}{\mathrm{N_p}} + \mathrm{T_S} - \mathrm{T_{PP}}} =$$

$$= \frac{1}{1 - \frac{T_{PP}}{T_S} + \frac{T_{PP}}{N_p \cdot T_S}} = \frac{1}{(1 - p) + \left(\frac{p}{N_p}\right)} \tag{4.1}$$

Where we have defined the *parallel content,* p, as a proportion:

$$p = \frac{\text{potentially parallel time}}{\text{whole code time}} \tag{4.2}$$

Expression (4.1) results in the well known Amdahl's law (Amdahl, 1967) which is applied to calculate the theoretical speed-up as a function of the parallel content p and the number of CPUs that will be used N_p. One important conclusion of this law is that the maximum speed-up accomplished is *finite*, that is, there is an upper bound for the speed-up that can be achieved by the parallel program, regardless of the number of processors used; when $N_p \to \infty$, the maximum achievable speed-up is $1/(1 - p)$.

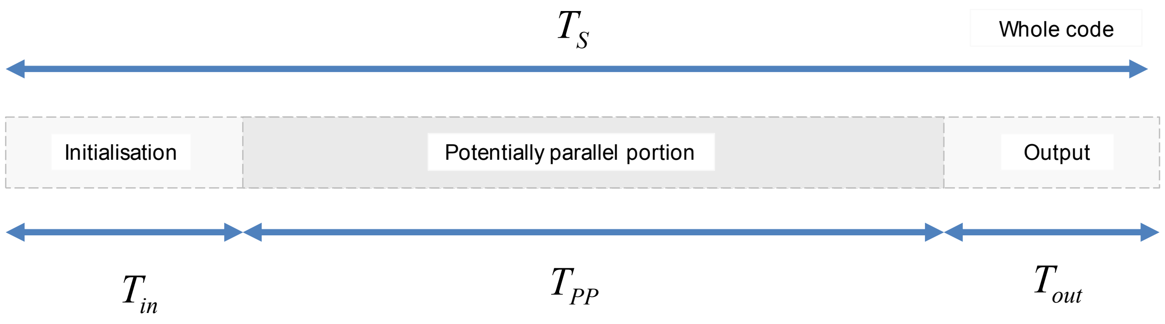


Fig. 42 Timing the baseline program to estimate likely parallel performance.

Let us incorporate a more realistic approach for the calculation of the maximum speed-up achievable. After the parallelisation of the nucleosynthesis at each of the shells, it is clear that each processing node will distribute to the other processors the mass fraction abundances obtained in the computation of their assigned shells. This represents an ALLGATHER communication procedure (MPI Forum, 2009) where all processors get the data sent by the rest of processing nodes. The information in this case is distributed by means of a *ring* algorithm where in the first step of the algorithm, each node i sends its contribution to node $i+1$ and receives the contribution from the processor $i-1$ (with wrap-around). From the second step onwards, each process i forwards to process $i+1$ the data it received from process $i-1$ in the previous step (Pacheco 1997). The time taken by this algorithm is given by (Thakur *et al*, 2002):

$$T_{comm} = (N_p - 1)\alpha + \frac{(N_p - 1)}{N_p} n\beta \tag{4.3}$$

where n is the total size of the data to be received by any process from all other processors, α is the latency (or start-up time) per message, independent of message size, and β is the transfer time per byte. With the incorporation of the communication time, the formula (4.1) can be expressed as:

$$\text{Speed} - \text{up} \approx \frac{1}{(1 - p) + \left(\frac{p}{N_p}\right) + \frac{T_{comm}}{T_S}} \tag{4.4}$$

It has to be noted that both the latency and the transfer time depend specifically on the speed of the network and of the communications of the cluster of processors where the parallel application is being executed. It will also depend on the heterogeneity of the nodes (e.g. workstations with different processing power, or different Operating System), and ultimately on how finely has been the cluster tuned to optimise data transfer and communications. They are therefore difficult to be analytically predicted, and are usually measured using real data and extrapolating communication times from observations (Foster, 1995).

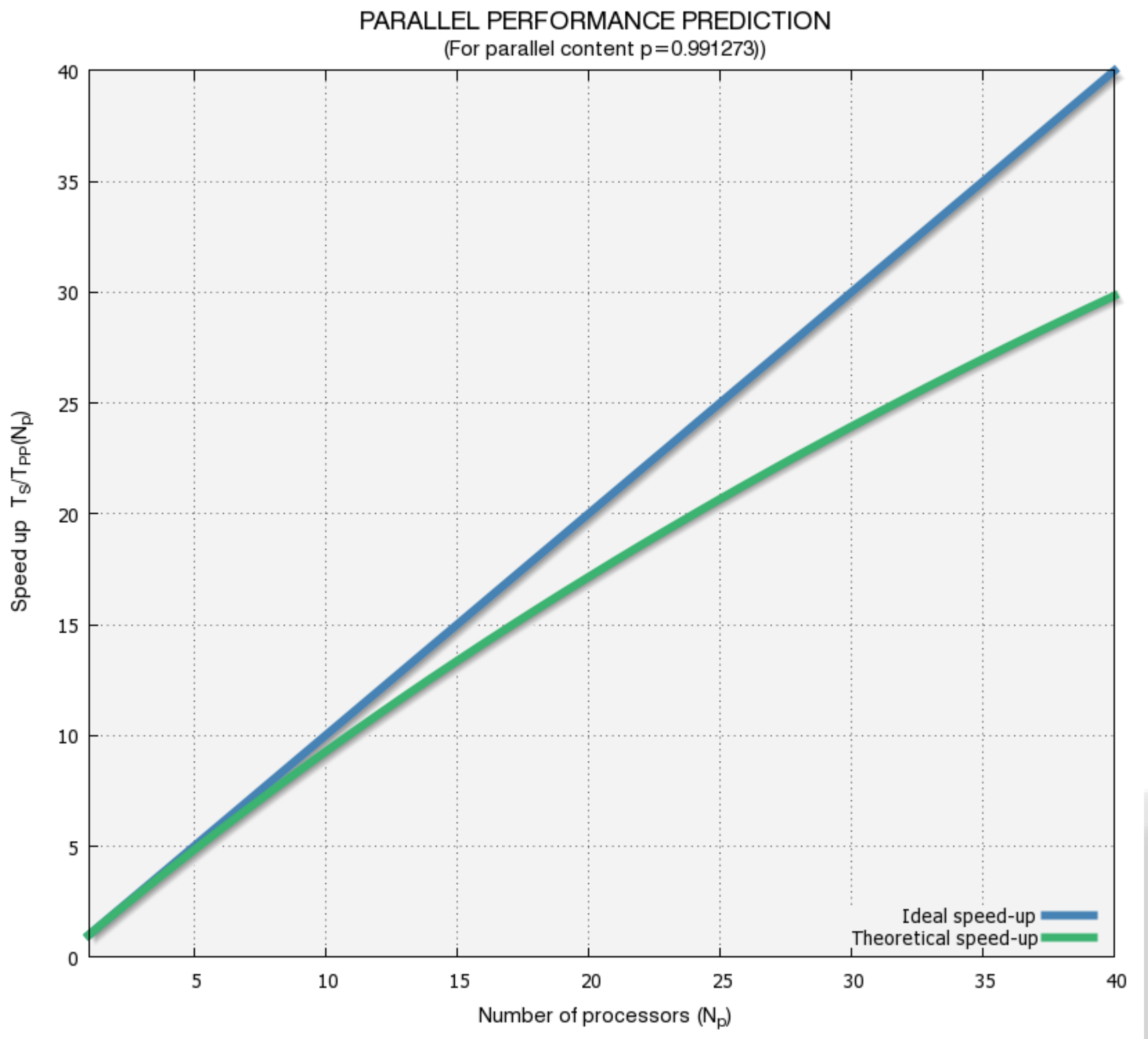


Fig. 43 Estimation of the parallel performance of the parallel version of the SHIVA code

Executions of the serial, baseline version of the SHIVA code, have yielded a parallel content coefficient of $p = 0.991273$. That means that the code spends the most part of the processing time calculating the energy and nucleosynthesis. This is extremely fortunate, since as it turns out, the processing stage that represents the main opportunity for parallelisation, is also where the code spends most of the execution time (more than 99% of the time!). This excellent result provides for very good theoretical speed-ups when used in formula (4.1), and definitely encourages the parallelisation of the SHIVA code using this strategy. This is depicted in Fig. 43, where the theoretical maximum achievable speed-up is depicted along with the ideal (unreachable) $\mathrm{N_p}$ (which reflects the idea that applying $\mathrm{N_p}$ CPUs to a program should cause it to complete $\mathrm{N_p}$ times faster). It can be seen that the theoretical speed-up differs more and more from ideal speed-up as the number of processors increases. This gap from the ideal to the theoretical speed-up is a function solely of the program‘s serial content. The theoretical estimation of the parallelisation of the SHIVA code yields nearly ideal speed-ups for $\mathrm{N_p} \leq 10$, and very good speed-ups for a larger number of processors. For instance, using all 44 processors of the Hyperion cluster (see section 2.5), a theoretical speed-up = 32 is expected. This is an excellent and promising result. We will see this result confirmed in the simulations, as shown in the following sections.

4.3.3. SHIVA Code Parallelisation

As discussed in the parallelisation analysis in the previous section, the main strategy to be adopted will be to parallelise only the nucleosynthesis and energy generation subroutines, as these are executed independently on each of the shells, therefore conforming to almost a perfect parallel problem architecture. The main design decision to be made here is what portion of the non-parallelisable code the nodes will execute. There are two main options to be adopted, namely:

- Option 1: Only process root executes all processing stages. When energy and nucleosynthesis are to be calculated for the current time-step, process root broadcasts to all other processors the necessary information to compute the nucleosynthesis for a subset of shells. After the work has been split amongst processors, each CPU can proceed forward with the computation, independently of the other processors. After all processors have finished the processing stage, process 0 gathers the new abundances and energy generated for all shells from all processors. Process root continues with the simulation (mass accretion, mass rezoning, extrapolation of new physical variables, etc) and goes on with a new iteration for a new converged model. In the meanwhile, all other processors remain idle, waiting for the process root to broadcast the input data for the energy and nucleosynthesis calculation of the next model.

- Option 2: All processors execute an instance of the SHIVA code, so that each process computes all processing stages (equations of matter, opacities and artificial viscosity, linearised system of equations for the physical values, solution of the system by means of the Henyey's method, etc.). When it comes to the computation of the nucleosynthesis and energy generated, each process performs the computation *only on a subset of all shells*. After this, each process broadcasts to the rest of the processors the results of the nucleosynthesis for their subset of shells. From here onwards, the simulation proceeds redundantly on all nodes, all of them executing the same code with the same input data.

In light of the results for the parallel content of the SHIVA code (that is, the nuclear subroutine consumes 99% of the computing time), is it safe to assume that the computation time devoted to all the other processing stages will be small, compared to the time needed in each iteration to broadcast to all processors the data needed to compute their subset of energy and nucleosynthesis. As a matter of fact, we experienced this behaviour in the parallelisation of the post-processing code (see section 3.5), where the communication time needed to distribute the system of equations amongst the processors, was significantly exceeding the time needed for one processor to solve the system of equations sequentially. A similar behaviour is expected here; given the small percentage of time spent on the rest of processing stages, it is assumed to be faster that each processor executes them sequentially (though redundantly) than to send and receive data back and forth at each iteration. Consequently, in this Thesis we have implemented Option 2. This choice has the added benefit that when the several processors arrive to the nucleosynthesis and energy generation processing stage, they all have *all information needed* to perform their computations (mainly temperature, density, and current abundances), and therefore only a broadcast of the new calculated abundances and energy released has to be executed afterwards.

The SHIVA code parallelisation strategy and processing stages are depicted in Fig. 44 overleaf. Vertical columns represent the several nodes participating in the parallel execution, and execution time flows downwards. Note that there are only two points of communication; once at the beginning of the simulation (where the root process broadcasts all initial information and parameters to all the processors), and subsequently at each iteration model, after the distributed computation of the nucleosynthesis has been done. Note that this design choice is in line with the principles discussed in section 2.3, where it was shown that in order to maximise parallel performance, the communication points had to be kept to a minimum or, in other words, the computation to communication ratio had to be maximised. In Fig. 44, all tasks spreading across the nodes are executed simultaneously by all processors. Note how all reading and writing tasks (initialisation and output phases) are executed by the root process only, which sends out the needed information to all processors. Information broadcast is kept to a minimum, by sending only information relevant to the nodes for their execution, that is, information relevant only to process 0 is not distributed (e.g. information only relevant to be written to a file).

In order to distribute equivalent workloads to all processors, the total number of shells of the neutron star envelope is split up into approximately equally sized groups. The shells assigned to each node are consecutive, so that CPUs compute energy and nucleosynthesis for shells $1...j$, $j+1...i$, $i+1...m$, and so on. Being N the number of all shells, the last processor will be assigned shells $m+1$ to N. It has to be pointed out that the current version of the SHIVA code maintains constant the number of shells (N) used in the simulation (with a required mass rezoning and physical variables interpolation). Be that as it may, it is envisaged for the near future a modification of the code so as to include a variable, increasing number of shells (hence not loosing *resolution* in the envelope as matter is accreted onto the neutron star in each iteration).

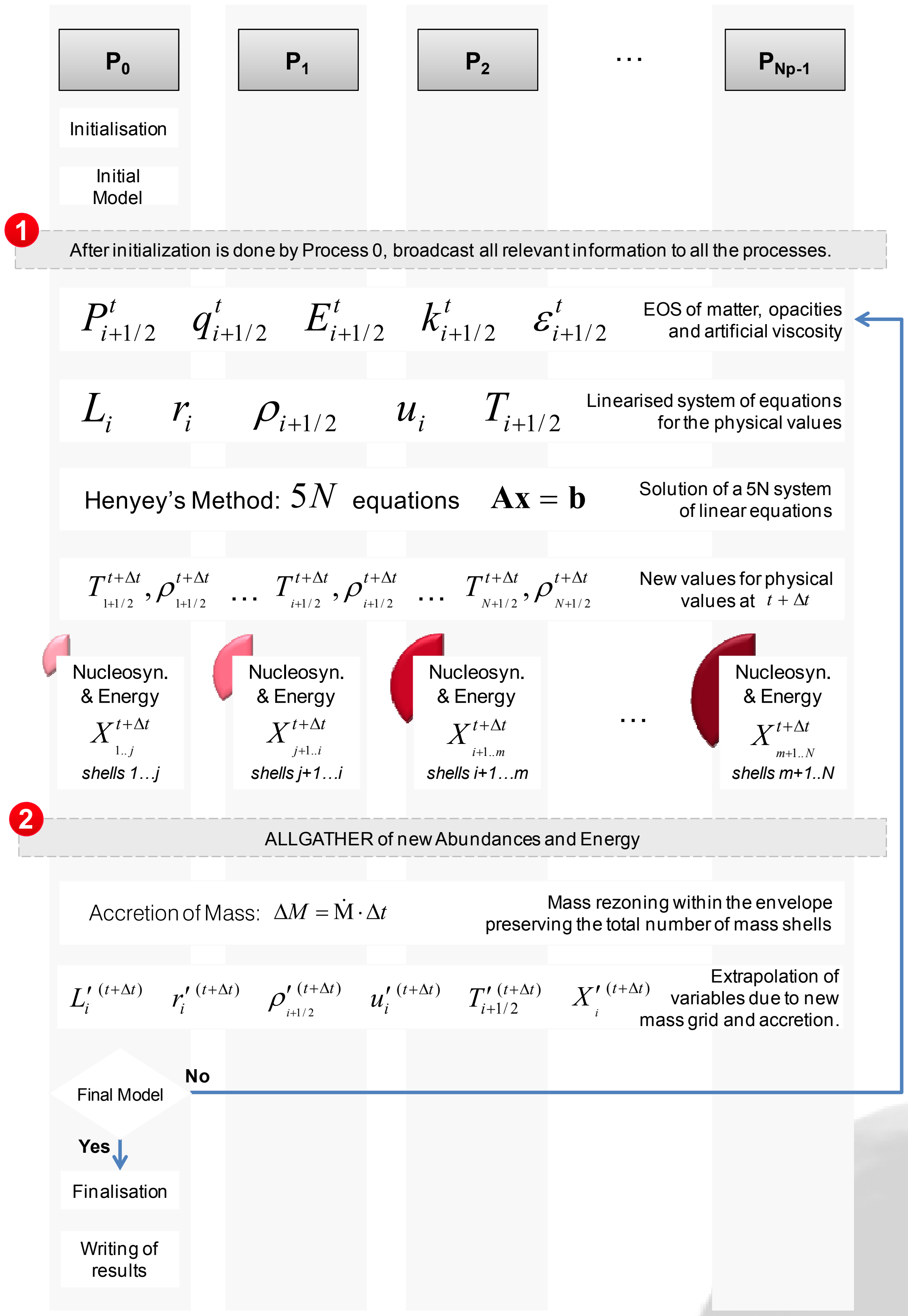

P_0
P_1
P_2
...
P_{Np-1}
Initialisation
Initial Model
1
After initialization is done by Process 0, broadcast all relevant information to all the processes.
$P^t_{i+1/2}$ $q^t_{i+1/2}$ $E^t_{i+1/2}$ $k^t_{i+1/2}$ $\varepsilon^t_{i+1/2}$
EOS of matter, opacities and artificial viscosity
L_i r_i $\rho_{i+1/2}$ u_i $T_{i+1/2}$
Linearised system of equations for the physical values
Henyey's Method: $5N$ equations $\mathbf{Ax} = \mathbf{b}$
Solution of a 5N system of linear equations
$T^{t+\Delta t}_{1+1/2}, \rho^{t+\Delta t}_{1+1/2}$ ... $T^{t+\Delta t}_{i+1/2}, \rho^{t+\Delta t}_{i+1/2}$ ... $T^{t+\Delta t}_{N+1/2}, \rho^{t+\Delta t}_{N+1/2}$
New values for physical values at $t + \Delta t$
Nucleosyn. & Energy $X^{t+\Delta t}_{1..j}$ shells 1…j
Nucleosyn. & Energy $X^{t+\Delta t}_{j+1..i}$ shells j+1…i
Nucleosyn. & Energy $X^{t+\Delta t}_{i+1..m}$ shells i+1…m
...
Nucleosyn. & Energy $X^{t+\Delta t}_{m+1..N}$ shells m+1..N
2
ALLGATHER of new Abundances and Energy
Accretion of Mass: $\Delta M = \dot{\mathrm{M}} \cdot \Delta t$
Mass rezoning within the envelope preserving the total number of mass shells
$L'^{(t+\Delta t)}_i$ $r'^{(t+\Delta t)}_i$ $\rho'^{(t+\Delta t)}_{i+1/2}$ $u'^{(t+\Delta t)}_i$ $T'^{(t+\Delta t)}_{i+1/2}$ $X'^{(t+\Delta t)}_i$
Extrapolation of variables due to new mass grid and accretion.
Final Model
No
Yes
Finalisation
Writing of results

Fig. 44 SHIVA code parallelisation strategy and processing stages

4.3.4. Validation of the Parallel Application

Contrary to the parallelisation of the post-processing nucleosynthesis code, the strategy adopted for the parallelisation of the hydrodynamic SHIVA code does not require to change the method of inversion of the nucleosynthesis matrix. In the parallel solution of the matrix arising from the linearisation of the set of differential equations describing the temporal evolution of the network abundances, it had to be used a parallel solver (MUMPS, see section 3.3.4), which employed a different method for matrix inversion than the Gaussian elimination technique used in the sequential version of the post-processing nucleosynthesis code. Since double precision figures are employed in the computation of the abundances evolution, this change in the solver implied that the results of the nuclear abundances did not tally with the results of the serial version to the very last decimal (although, it was obviously very close). This required to set up a validation procedure (3.4 - Validation of the Parallel Application) to ascertain that the parallel version of the code actually yielded the same results as its sequential counterpart.

This procedure is not necessary in the case of the parallelisation of the hydrodynamic SHIVA code. The stages being parallelised are still being executed *sequentially* on each of the nodes, so the methods of matrix inversion and nucleosynthesis calculation do not change. As a result, all output files resulting from the parallel execution have been found to be verbatim with respect to the output files generated with the sequential execution. The validation has been therefore carried out with a simple file difference comparator utility.

4.4. Results and Discussion

The theoretical performance predicted in section 4.3.2 has been successfully confirmed by the simulations with the parallelised version of the SHIVA code. Fig. 45 shows the excellent results of the speed-up factors accomplished in a simulation with 100 time-steps and N=200 shells. Note that 100 time-steps constitute a very limited hydrodynamic simulation (usually simulations can be run for about 200.000 time-steps), but it is representative enough to calculate parallel execution times with respect to a serial execution with a single processor. Simulations have been carried out with two different nuclear reaction networks; one with a reduced network consisting of 324 isotopes and 1392 reactions, and another one with a far more complete reaction network up to 606 nuclides and 3551 nuclear reactions (Moreno 2009).

	Number of shells (N)	Number of computed time-steps	Nuclides	Nuclear Reactions
Reduced Simulation	200	100	324	1392
Extended Simulation	200	100	606	3551

Table 3 Simulations run for the performance evaluation of the parallel SHIVA code

Results of the reduced and extended simulations are shown in Fig. 45. It can be seen that a speed-up factor of **26** is achieved with the reduced simulation when 42 processors are used in parallel to execute the application. On the other hand, an *excellent* speed-up factor of **35** is accomplished with the extended simulation when all 42 processors are used in the simulation. Note that even though the Hyperion cluster is formed by 44 CPUs, there was a non operative node that reduced the amount of available processors to distribute parallel work to. The results obtained are so good that stop short of the results that could be obtained with a perfect parallel application; this means that the computation to communication ratio is large enough so that processing work can be distributed in an extremely efficient way amongst processors. As it was discussed in section 4.3.3, a parallelisation strategy has been adopted, so that the global communications have been reduced to a single point of information exchange per iteration. Consequently, each node has a significant amount of processing work to complete before they need to communicate with the rest of the processors. This leads to a considerable improvement of the computation speed.

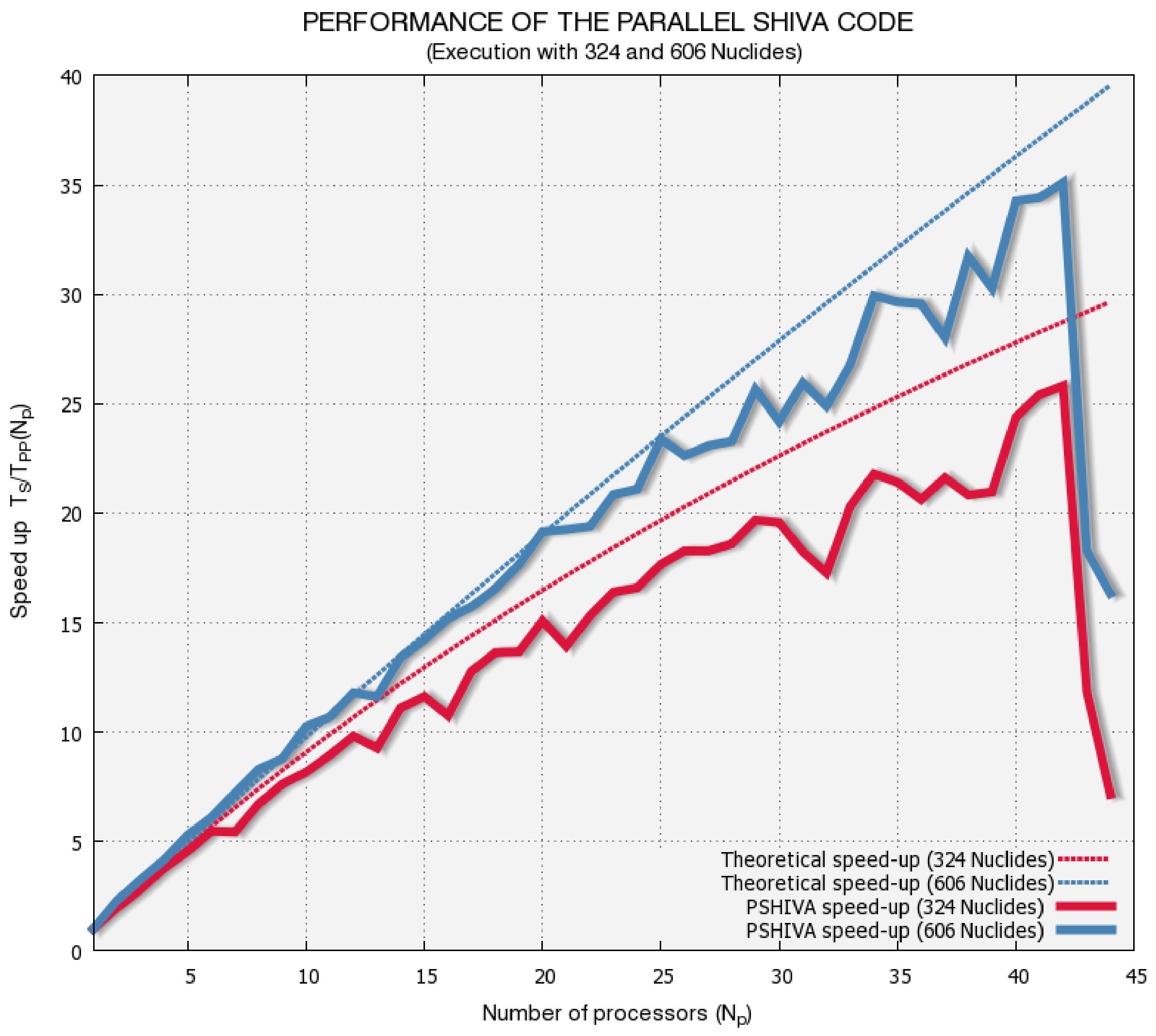


Fig. 45 Performance of the parallel SHIVA code for executions with 324 and 606 nuclides

In Fig. 45 there have also been included the *theoretical* speed-ups for both simulations. These theoretical estimations do not take into account the communication or synchronisation times (see section 4.3.2), as a result, the observed performance will always fall below the theoretical, ideal speed-up. Overheads also have an impact on the execution time and contribute to the deviation of the observed speed-up from the theoretical speed-up. This overhead stems mainly from two sources, both of them out of our control: on the one hand the additional CPU cycles devoted simply to the management of the parallelism, and the wasted time or delays spent waiting for communications to complete. On the other hand, competition from the operating system or even other users using the cluster also can affect performance to a certain degree. The reason for the theoretical speed-up not taking into account these factors is that it would overcomplicate the model to predict performance of the parallel application. Finally, some minima can be seen in the figure (e.g. 32 and 37 processors). They are caused by an uneven distribution of workload for this specific number of processors. It is also possible that the processors used at those points are connected with slower Ethernet connections to the clusters, or that perform additional cluster management tasks that uses up part of their computational resources.

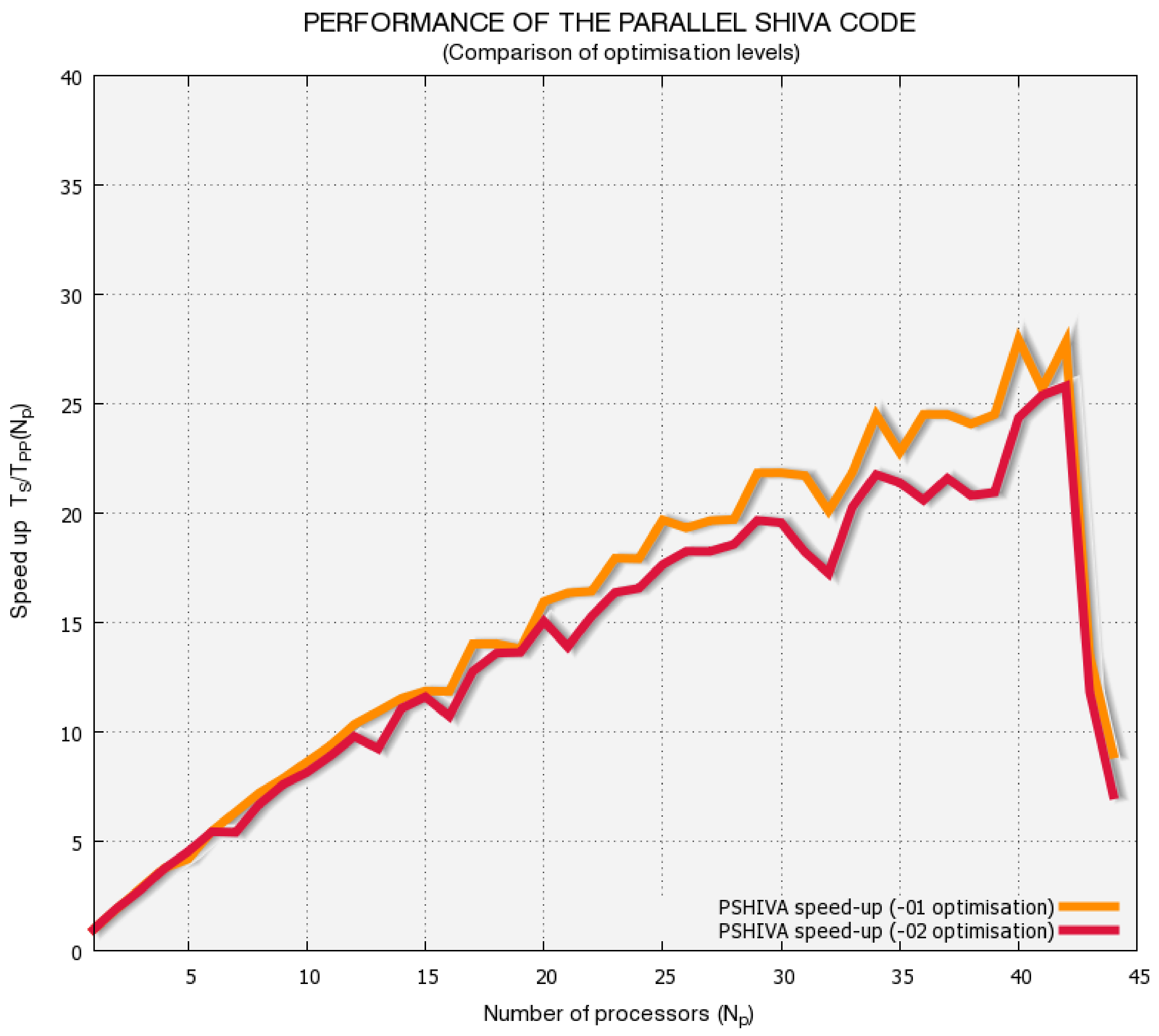


Fig. 46 Performance of the parallel SHIVA code for different levels of compiler optimisation

Note the loss in performance when more processors are used than the physically available CPUs (42 effective CPUs). For $N_P=43$, and $N_P=44$, approximately 100% and 200% worse execution times are obtained respectively, as compared to the execution times of the parallel application when using 42 processors. The fact that a single CPU has to interleave the work of more than one node, introduces interruption, synchronisation, and prioritisation overheads that extremely penalise the concurrent execution of the simulation. The maximum number of physically available CPUs will constitute therefore an effective limit in the number of processors used in the parallel execution.

As expected, considerably higher speed-ups are obtained when we increase the problem size by using a nuclear reaction network with 606 isotopes and 3551 reactions. The speed-up accomplished in this simulation exceeds in approximately 34% the performance of the execution with a reduced nuclear network (26 versus 35 factors respectively). This is a consequence of increasing the problem size, which is essentially equivalent to increasing the amount of parallelisable computation (that is, the nucleosynthesis calculation), and therefore the potential parallel content also increases ($p = 0.991273$ for the reduced simulation, whereas $p = 0.997382$ for the simulation with an extended nuclear reaction network). This, in turn, improves the curve of the modelled, theoretical speed-up, hence diminishing the gap from ideal speed-up.

The effect of increasing the amount of parallelisable computation also occurs when different optimisation options are used in the compilation of the parallel application code. Fig. 46 shows the speed-up obtained for a simulation using a reduced network of 324 isotopes and 1392 reactions, with compiler optimisation options `-01` and `-02`, respectively. Option `-01` omits optimisations that tend to increase the object size, and creates smallest optimised code. On the other hand, option `-02` (the default setting) enables many optimizations, including vectorisation and creates a faster execution code than option `-01` (Intel 2011). The code with the smallest level of optimisation executes slower than the code compiled with a faster optimisation option. This has the effect of increasing the computation time (while the communication times are maintained), and therefore the speed-up accomplished increases. It is important to remark that it is the *speed-up* that is increased because of the slower computation times, but the total *execution time* is worse for the code compiled with option `-01` than for the code compiled with option `-02`. As an example, the default optimisation level (`-02`) runs the code with a single node in 264.3 seconds, whereas if no optimisation is used (`-01`), the sequential version of the code completes in 395.9 seconds (it takes a 50% longer to complete). It is obviously preferable to use the default optimisation option to compile the code.

100 Time-steps execution	$N_P = 1$	$N_P = 10$	$N_P = 20$	$N_P = 42$
Reduced Simulation (324 isotopes, 1392 reactions)	264.3 sec.	32.3 sec.	17.5 sec.	10.2 sec.
Extended Simulation (606 isotopes, 3551 reactions)	1237.8 sec.	120.6 sec.	64.5 sec.	35.2 sec.
Ratio	4.7	3.7	3.6	3.4

Table 4 Execution times for the reduced and extended simulations for different number of processors

In line with the above, execution times of the simulation with 324 and 606 nuclides are depicted in Fig. 47, where execution times have been normalised to the sequential, single node execution times of the simulation with the reduced nuclear reaction network. As the number of nodes increases, execution time decreases faster for the extended simulation than for the reduced one (i.e. larger speed-ups are accomplished, as shown in Fig. 45), but it obviously takes more time to complete the simulation since the amount of data to be processed is substantially larger. It is interesting to see that the ratio between the execution times for the reduced and extended simulations, decreases almost monotonically, albeit slowly, as the number of CPUs participating in the execution is augmented. This tendency brings us to think that for a sufficiently high number of parallel processors, the ratio of execution times for the reduced and extended simulation may stabilise and converge to a specific, constant value. Table 4 lists explicitly the execution times for the reduced and extended nuclear reaction networks in a simulation with 100 time-steps. The ratio between execution times decrease from 4.7 with $N_P=1$, down to 3.4 with $N_P=42$.

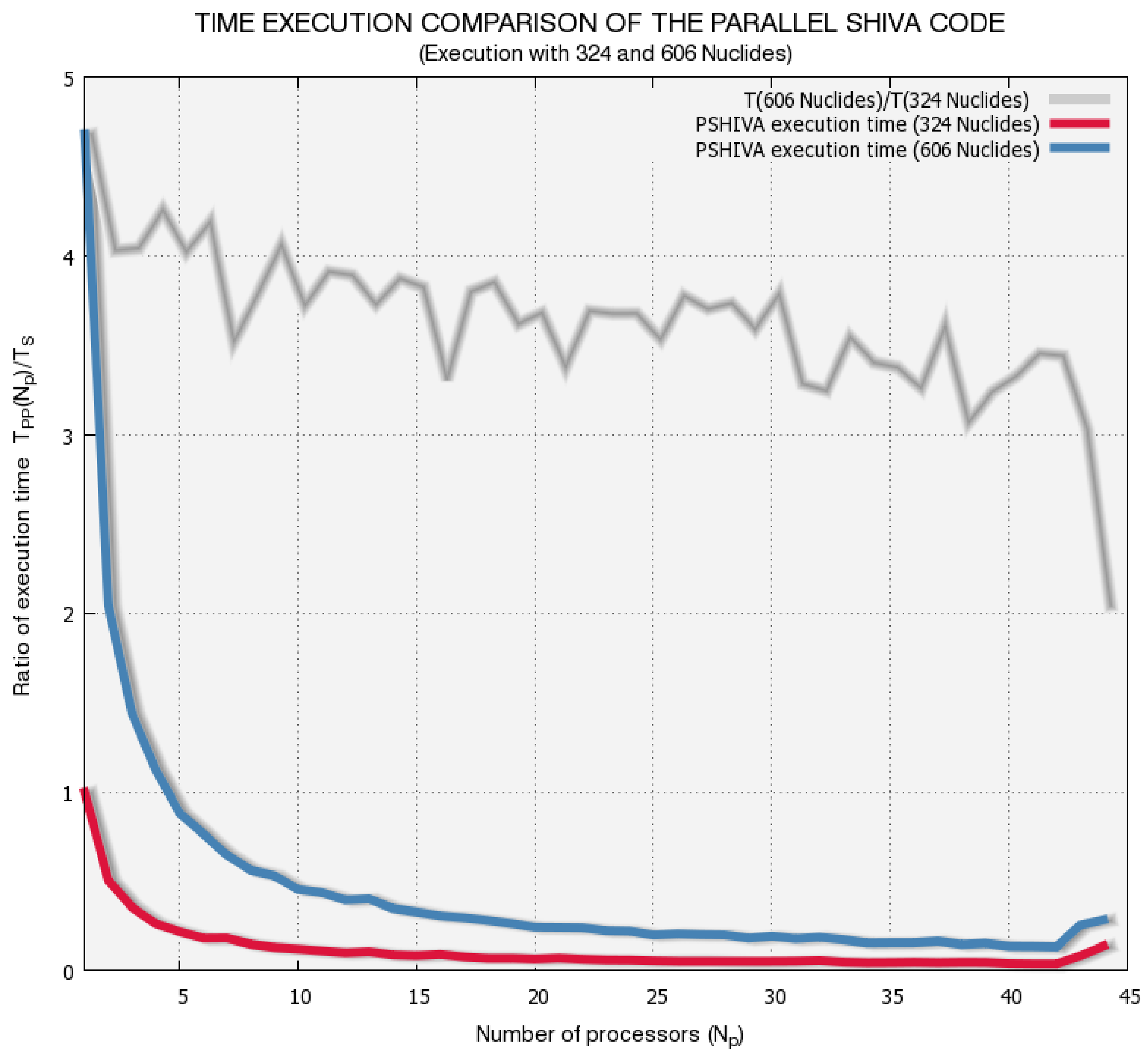


Fig. 47 Ratio of execution time of the parallel SHIVA code with 324 and 606 nuclides

Amdahl's law (Amdahl, 1967) was introduced in section 4.3.2 and was shown to predict an upper limit to the potential speed-up that can be accomplished by applying multiple CPUs to the parallel solution of a complex problem. It was argued that this upper limit depends on the amount of code that cannot be parallelised (the so-called, serial content of the program) and that was independent of the number of processors being used in the simulation. A number of authors have discussed the relevance of the Amdahl's law arguing that it is possible for N_P processors to execute a program in less than 1/ N_P of the time that it takes to execute serially (see for instance Venkatesh *et al.* 2005, Rao *et al.* 1998, and Sutter 2008). This is the so called *superlinear speed-up* and it is mainly attributable mainly to differences in the parallel and serial versions of the code, secondly to cache optimisation differences when more than one processor are used (e.g. a better use of cache memory), and finally to small differences in the initialisation or output phases of the execution. In our simulations we have achieved superlinear speed-ups when executing a simulation with 606 isotopes and 3551 nuclear reactions network, running in parallel with less than 10 processing nodes. Details of the speed-up shown in Fig. 45 are represented in Fig. 48 at a larger scale for a maximum of 6 CPUs. It is surprising that superlinear speed-ups are accomplished, for instance for N_P=2, 3, 4 ,5 and 6. For instance, the execution of the simulation with 3 processors completes 3.25 times faster than the sequential, one-node execution.

The reason for this surprising result is attributable to two different facts. On the one hand there is a certain degree of standard deviation in the measurement of the execution times accomplished throughout the simulation. Small variations in the synchronisations, overheads, interruptions from the operating system, and even other users connecting to the cluster provoke small deviations in the measured execution times from one simulation the other. This causes that even if the number of CPU is kept constant, different executions may yield slightly different execution times. Even though the deviation is expected to be small when compared to its expected value, this fact adds a *noise* in the precise measurement of execution times. It must be stressed that implementing the appropriate procedure to compute the speed-up would require taking a statistically significant number of samples (for a fixed number of CPUs), and computing afterwards the mean value. This is not necessary, as the uncertainty is considered to be small enough so as not to affect substantially the result.

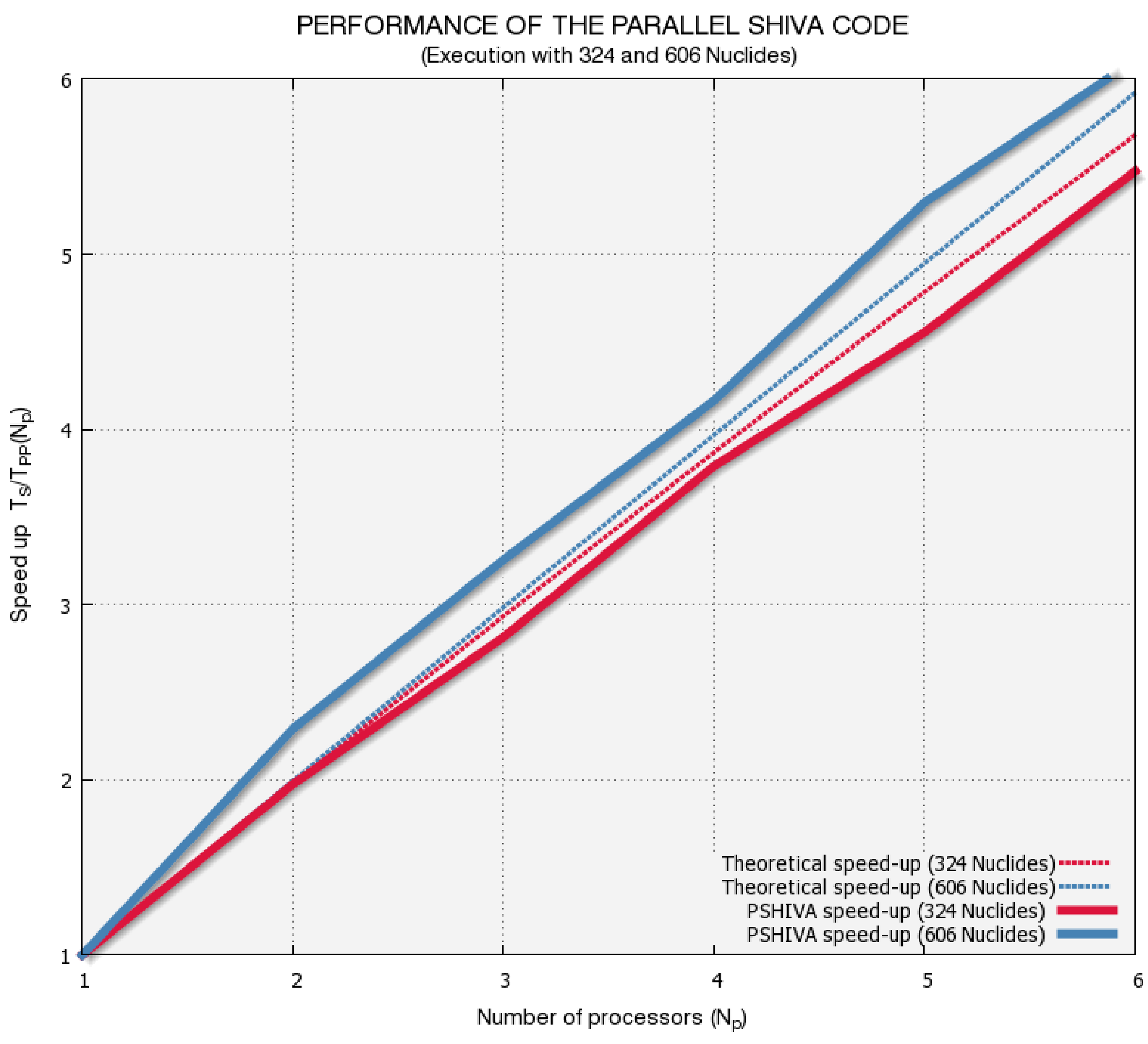


Fig. 48 Performance of the parallel SHIVA code. Detail for a reduced number of processors

Having ruled out the aforementioned imprecision in the measurement of execution times, other causes have to be contemplated as the main reason for this superlinear behaviour. One of the main reasons for superlinear speed-ups is the differences in the serial and parallel versions of the code. In section 3.3.2 it was described an optimisation implemented as part of this Thesis, aimed at improving the computational efficiency of the interpolation of the reaction rates. Additionally, with the aim of improving efficiency, partial values accumulators have been used in the construction of the matrix $\mathbf{A}$ that arises from the linearisation of the set of differential equations describing the temporal evolution of the network abundances. Also, it is conceivable that the smaller computational workload that each CPU has to cope with, generates a more efficient use of the processor resources (for instance, cache memory). All these factors contribute to an increased efficiency and reduced execution times when executing the parallel application. Note that this superlinear behaviour is not obtained in the parallel execution of the reduced simulation with 324 isotopes and 1392 nuclear reactions network. The computational intensity in this simulation is significantly smaller in this case and the benefits of all these optimisations much less noticeable.

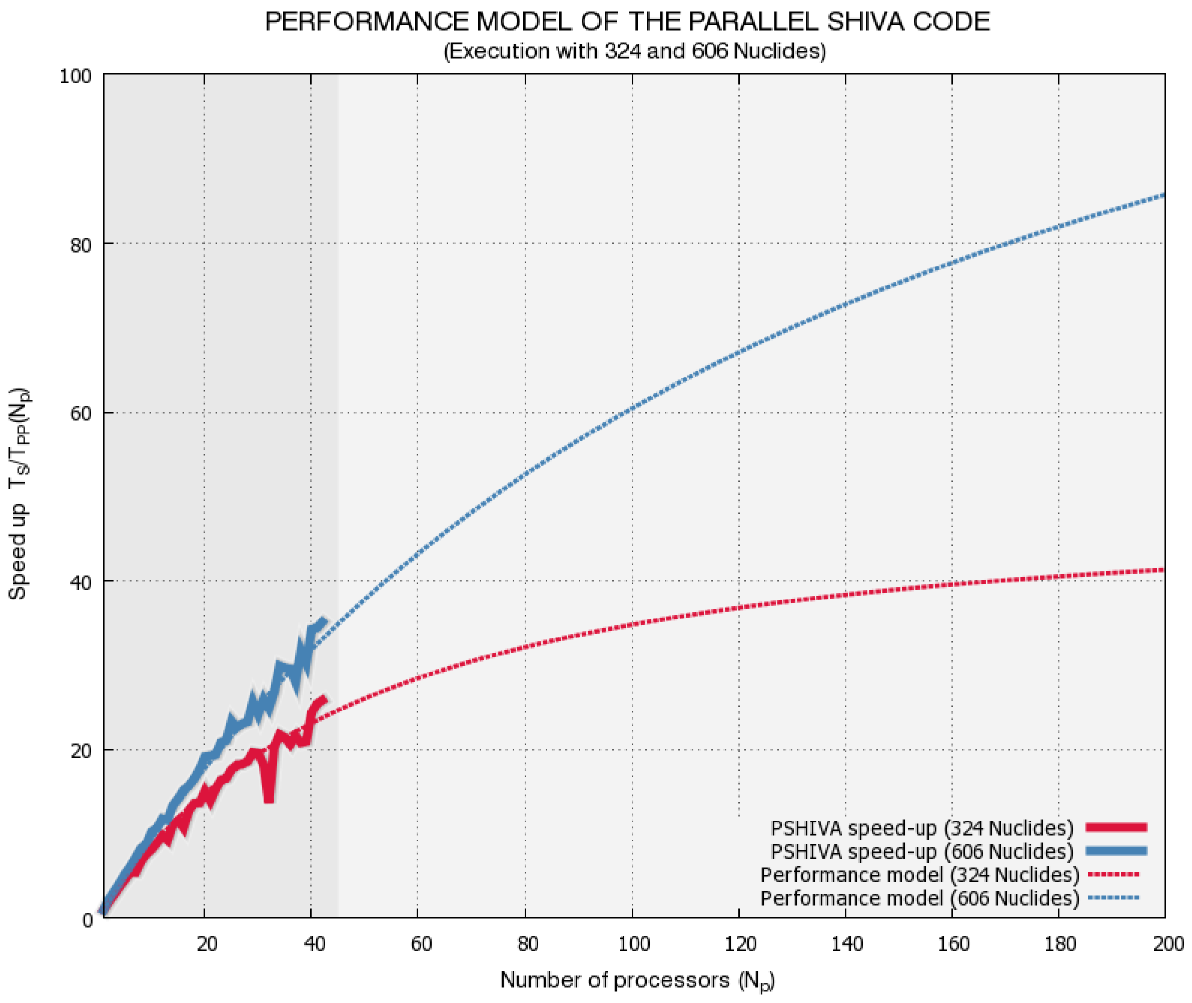


Fig. 49 Performance model of the parallel SHIVA code up to 200 processors

From equations (4.3) and (4.4), the model of the performance of the parallelised SHIVA code, incorporating the communication time between nodes, can be expressed as:

$$\text{Speed}-\text{up} \approx \frac{1}{(1-\text{p}) + \left(\frac{\text{p}}{\text{N}_\text{p}}\right) + \frac{\left((\text{N}_\text{p}-1)\alpha + \frac{(\text{N}_\text{p}-1)}{\text{N}_\text{p}} n\beta\right)}{\text{T}_\text{S}}} \tag{4.5}$$

where n and T_S will depend on the simulation being executed, whereas the latency α , and the transfer time per byte β will depend solely on the cluster communications network where the simulations are being executed. Experimental measures in the GAA's Hyperion cluster have yielded the values $\alpha = 1\cdot10^{-5}$ and $\beta = 5\cdot10^{-8}$. At the end of every iteration, all nodes have to receive the result of the nucleosynthesis for all shells, with the addition of the total nuclear energy generated and the new calculated Δt at each of the shells. Taking all this into account the total amount of bytes being transmitted works out at:

n =200 shells x (324 isotopes x 8 bytes/isotope +
8 bytes/shell (energy) + 8 bytes/shell (Δt)) = 521.6 kb

n =200 shells x (606 isotopes x 8 bytes/isotope +
8 bytes/shell (energy) + 8 bytes/shell (Δt)) = 972.8 kb

respectively for the reduced (324 isotopes) and extended (606 isotopes) simulations.

The performance model of the parallel SHIVA code (eq. 4.5) is depicted in Fig. 49 up to 200 processors. We have also included the experimental data obtained up to N_P=42 processors in the Hyperion cluster. It is interesting to note that there is still way for improvement and that the maximum is not reached even when using all 42 available CPUs of the Hyperion cluster. Maximum speed-ups of ~41 and ~85 are predicted by the model when using 200 processors, for the reduced and extended simulations respectively. At this point it is important to be aware that as a result of the parallelisation strategy that has been adopted (see Fig. 44), the number of shells of the neutron star envelope constitute an effective *upper limit* for the maximum number of CPUs that could be used in the parallel application. Any CPUs used above that limit would be literally a wasted resource, since there would be no shells left for those processors to work with. It must be stressed here that splitting the nucleosynthesis calculation of a single shell amongst several processors is not a viable alternative; as it has been shown in section 3.5; the nucleosynthesis computation stage constitutes a loosely coupled application that cannot be parallelised.

The model of performance presented is valid *on the Hyperion cluster*, and cannot be extrapolated to other cluster which may have different values for latency and communication bandwidths. However this model can be taken as a reference for the capabilities of the parallelised application. For instance, based on the above presented results, we may decide on making an application for computation time at some supercomputing facility (e.g. at the Mare Nostrum supercomputer at the Supercomputing Centre in Barcelona (BSC)), where latencies and transmission bandwidth are highly optimised for parallel executions. In this case, even better results should be expected than those presented here. Over and above, we would be able to use as many processors as there are shells in the neutron star envelope, thus reaching the maximum speed-up attainable with the parallel SHIVA code.

CHAPTER 5
SUMMARY AND CONCLUSIONS

"It is not the possession of truth, but the success which attends the seeking after it, that enriches the seeker and brings happiness to him."

Max Planck (1858-1947)
German physicist. Nobel prize for physics, 1918.

In this Master Thesis, two numerical codes have been parallelised using the MPICH2 implementation of the Message Passing Interface (MPI) specification for the design of parallel applications with clusters of distributed workstations. The first application being parallelised has been a nucleosynthesis code suitable for extensive post-processing calculations, with a network containing 606 nuclides (H to ^{113}Xe) and more than 3500 nuclear reactions (Moreno, 2009). This code requires (in its sequential, single-node version) about 9.1 CPU-months of calculating power to perform a sensitivity study of 50.000 post-processing calculations of X-ray bursts nucleosynthesis (Chapter 3). The second application is the hydrodynamic code SHIVA, a one-dimensional (spherically symmetric), hydrodynamic code, in Lagrangian formulation, built originally to model classical nova outbursts (José 1996; José & Hernanz 1998). A partial hydrodynamic simulation takes 147 hours (6.1 days) to run for 200k time-steps using a reduced nuclear network with 324 isotopes and 1392 nuclear reactions. The computation time soars to 688 hours (28.6 days) when using a network with 606 nuclides and 3551 nuclear reactions, with the same number of time-steps.

The main goal of the of the parallelisation has been to benefit from the 42-node Hyperion Cluster that the Astronomy and Astrophysics Group (GAA) has at the EUETIB (UPC), hence achieving significant speed-ups in the simulations. As a consequence, faster computations will pave the way for better numerical approaches and finer approximations (e.g. more isotopes and reactions, or more layers of the neutron star envelope's model) that were previously prohibitive due to its high computational requirements. Another side effect is that GAA's simulations and publications will benefit from a shorter time-to-publication, resulting from simulations running faster with the parallelised application. With the simulation code being parallelised, it will also be possible to take advantage, of parallel supercomputing facilities like the *Mare Nostrum* at the Supercomputing Centre in Barcelona (BSC), for the most demanding simulations.

The time dependent iterations of the nucleosynthesis post-processing code, places this application in the worst possible category for parallelisation (a loosely synchronous application), where all processors have to participate throughout the iteration, exchanging intermediate results in a regular basis. Also, the resulting abundances have to be broadcasted to all processors at the end of the iteration, so that they are readily available to every node at the following time-step for the distributed construction of the system of equations describing the temporal evolution of the network abundances. This is a serious bottleneck that provokes that the simulation cannot proceed until all processors have received the results. The post-processing nucleosynthesis code is a time-step loosely synchronous application with a very small problem size (limited by the number of isotopes of the nuclear network). It is therefore the worst possible scenario for parallelisation. As results have shown out, the performance of the parallel application is much worst than the sequential, 1-node version of the code. This stems from the fact that the communication and message passing times between processors largely outgrow the computation time. It is therefore not possible to parallelise efficiently a post-processing nucleosynthesis code, and efforts in this regard should be avoided.

All the contrary, the parallelised version of the SHIVA code shows excellent performance results, with significant speed-up factors accomplished in a simulation with N=200 shells. Simulations have been carried out with two different nuclear reaction networks; one with a reduced network consisting of 324 isotopes and 1392 reactions, and another one with a far more complete reaction network up to 606 nuclides and 3551 nuclear reactions (Moreno 2009). A speed-up factor of 26 is achieved with the reduced simulation when 42 processors are used in parallel to execute the application. On the other hand, an *excellent* speed-up factor of 35 is accomplished with the extended simulation when all 42 processors are used in the simulation. The results obtained are so good that stop short of the results that could be obtained with a perfect parallel application; this means that the computation to communication ratio is large enough so that processing work can be distributed in an extremely efficient way amongst processors. The parallelisation of the code has been realised in a way so as to reduce the global communications to a single point of information exchange per iteration. Consequently, each node has a significant amount of processing work to complete before they need to communicate with the rest of the processors. This leads to a considerable improvement of the computation speed. A parallel hydrodynamic simulation using 42 nodes, takes 5 hours and 39 minutes to run for 200k time-steps when using a reduced nuclear network with 324 isotopes and 1392 nuclear reactions. The computation time goes up to 19 hours and 40 minutes when using a network with 606 nuclides and 3551 nuclear reactions for the same number of time-steps. These are excellent results that completely justify the time invested in the parallelisation of the hydrodynamic simulation code. It is interesting to note that there is still way for improvement and that the maximum is not reached even when using all 42 available CPUs of the Hyperion cluster. Maximum speed-ups of ~41 and ~85 are predicted by the performance model when using 200 processors, for the reduced and extended simulations respectively.

The scope of the Work presented in this Master Thesis is planned to be extended in the forthcoming future, maybe by a PhD Thesis with special emphasis in the following aspects:

- Study of the dependence of XRB properties on the M-R relation obtained with different EOS for the neutron star interior.
- Characterization of XRB properties in primordial stellar binaries.
- Modify the SHIVA code to the study of superbursts.
- Improvement of the accretion procedure (from a fixed number of shells to an increasing number of shells as matter is accreted onto the envelope of the neutron star).
- Incorporation of rotation and other phenomena.
- Inclusion of general relativity corrections to the equations of stellar structure (Ayasli & Joss 1982), or transformation of the SHIVA code into a fully relativistic hydrocode (May & White 1967).
- Multidimensional studies of point-like ignition and flame propagation in the envelopes of accreting neutron stars.

APPENDIX A
SOFTWARE TOOLS

The following sections describe the software tools that have been used in the development of this Master Thesis. Only a brief description of functionality and availability is provided. In order to obtain build, compilation, and utilisation information please refer in each case to the indicated web pages.

A.1. MUMPS

MUMPS ("Multifrontal Massively Parallel Solver") is a package for solving systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a square sparse matrix that can be either asymmetric, symmetric positive definite, or general symmetric. MUMPS implements a direct method based on a multifrontal approach which performs a direct factorization

$$\mathbf{A} = \mathbf{LU} \tag{1}$$

where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ an upper triangular matrix. If the matrix is symmetric then the factorization

$$\mathbf{A} = \mathbf{LDL}^{\mathrm{T}} \tag{2}$$

where $\mathbf{D}$ is a block diagonal matrix with blocks of order 1 or 2 on the diagonal is performed. MUMPS exploits both parallelism arising from sparsity in the matrix $\mathbf{A}$ and from dense factorizations kernels.

Source: http://graal.ens-lyon.fr/MUMPS/index.php?page=doc

A.2. GotoBLAS2

The GotoBLAS codes are currently one of the fastest implementations of the Basic Linear Algebra Subroutines (BLAS). GotoBLAS2 uses new algorithms and memory techniques for optimal performance of the BLAS routines. The BLAS routines and functions are divided into the following groups according to the operations they perform:

- BLAS Level 1 Routines perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
- BLAS Level 2 Routines perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
- BLAS Level 3 Routines perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Source: http://www.tacc.utexas.edu/tacc-projects/gotoblas2

A.3. BLACS

The BLACS (Basic Linear Algebra Communication Subprograms) project is an ongoing investigation whose purpose is to create a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The length of time required to implement efficient distributed memory algorithms makes it impractical to rewrite programs for every new parallel machine. The BLACS exist in order to make linear algebra applications both easier to program and more portable. It is for this reason that the BLACS are used as the communication layer of ScaLAPACK (see next section). Key ideas in the BLACS include:

- Standard interface,
- Process grid and scoped operations,
- Contexts,
- Array-based communication,
- ID-less communication.

Source: http://www.netlib.org/blacs/

A.4. ScaLAPACK

The ScaLAPACK (or Scalable LAPACK) library includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers. It is currently written in a Single-Program-Multiple-Data style using explicit message passing for interprocessor communication. It assumes matrices are laid out in a two-dimensional block cyclic decomposition.

Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. (For such machines, the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and local memory on each processor.) The fundamental building blocks of the ScaLAPACK library are distributed memory versions (PBLAS) of the Level 1, 2 and 3 BLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the PBLAS and the BLACS. One of the design goals of ScaLAPACK was to have the ScaLAPACK routines resemble their LAPACK equivalents as much as possible.

Source: http://www.netlib.org/scalapack/

A.5. LAPACK

LAPACK (Linear Algebra PACKage) is written in Fortran 90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

The original goal of the LAPACK project was to make the widely used EISPACK and LINPACK libraries run efficiently on shared-memory vector and parallel processors. On these machines, LINPACK and EISPACK are inefficient because their memory access patterns disregard the multi-layered memory hierarchies of the machines, thereby spending too much time moving data instead of doing useful floating-point operations. LAPACK addresses this problem by reorganizing the algorithms to use block matrix operations, such as matrix multiplication, in the innermost loops. These block operations can be optimized for each architecture to account for the memory hierarchy, and so provide a transportable way to achieve high efficiency on diverse modern machines. We use the term "transportable" instead of "portable" because, for fastest possible performance, LAPACK requires that highly optimized block matrix operations be already implemented on each machine.

Source: http://www.netlib.org/lapack/

A.6. METIS/ParMETIS

METIS is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices. The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes.

ParMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. ParMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel AMR computations and large scale numerical simulations. The algorithms implemented in ParMETIS are based on the parallel multilevel k-way graph-partitioning, adaptive repartitioning, and parallel multi-constrained partitioning schemes.

Source (METIS): http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
Source (ParMETIS): http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

A.7. MPICH2

MPICH2 is a freely available, portable implementation of MPI, the Standard for message-passing libraries. It implements MPI-1, MPI-2, MPI-2.1 and MPI-2.2. MPI (Message-Passing Interface) is a message-passing library interface specification. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. (Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O.) MPI is a specification, not an implementation; there are multiple implementations of MPI. This specification is for a library interface; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, Fortran-77, and Fortran-95, are part of the MPI standard. The standard has been defined through an open process by a community of parallel computing vendors, computer scientists, and application developers.

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message-passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message-passing standard, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

The goal of the Message-Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

A complete list of goals follows:

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying, allow overlap of computation and communication, and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C, C++, Fortran-77, and Fortran-95 bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on many vendors' platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread safety.

Source (MPI): http://www.mcs.anl.gov/research/projects/mpi/
Source (MPICH2): http://www.mcs.anl.gov/research/projects/mpich2/

A.8. SCOTCH/PT-SCOTCH

SCOTCH and PT-SCOTCH are software package and libraries for sequential and parallel graph partitioning, static mapping, and sparse matrix block ordering, and sequential mesh and hyper-graph partitioning. Its purpose is to apply graph theory, with a divide and conquer approach, to scientific computing problems such as graph and mesh partitioning, static mapping, and sparse matrix ordering, in application domains ranging from structural mechanics to operating systems or bio-chemistry.

The SCOTCH distribution is a set of programs and libraries which implement the static mapping and sparse matrix reordering algorithms developed within the SCOTCH project. PT-SCOTCH is a parallel version of SCOTCH that makes use of the MPI interface.

Source: http://www.labri.fr/perso/pelegrin/scotch/

BIBLIOGRAPHY

Amdahl, G., 1967, *Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities* Proc. AFIPS Conf., p 483

Amestoy, P. R., Duff, I. S., L'Excellent, J.–Y., and Koster, J., 2000, *MUMPS: a general purpose distributed memory sparse solver.* Proceedings of PARA2000, Fifth International Workshop on Applied Parallel Computing, Bergen, June 18-21, p 122

Amestoy P. R., Duff, I. S., and L'Excellent, J.–Y., and Li, X., 2001a, *Performance and tuning of two distributed memory sparse solvers,* Proceedings of Tenth SIAM Conference on Parallel Processing for Scientific Computing. Norfolk, Virginia from March 12th-14th

Amestoy, P. R., Duff, I. S., Koster, J., and L'Excellent, J.-Y., 2001b, *A fully asynchronous multifrontal solver using distributed dynamic scheduling,* SIAM Journal of Matrix Analysis and Applications 23, No 1, p 15

Amestoy, P. R., Guermouche, A., L'Excellent, J.-Y., and Pralet, S., 2006, *Hybrid scheduling for the parallel solution of linear systems,* Parallel Computing 32 (2), p 136

Åström J., 2009, *Solving linear sets of equations for fusion plasma codes*, CSC magazine archive

Audi, G., Wapstra, A. H., & Thibault, C., 2003, Nuclear Physics A 729, p 337

Ayasli, S., & Joss, P. C., 1982, ApJ 256, p 637

Belian, R. D., Conner, J. P., & Evans, W. D., 1976, ApJ 206, L135

Bhattacharyya, S., 2006, *What Thermonuclear X-ray Bursts can tell us about Neutron Stars,* http://www.iiap.res.in/PostDocuments/SudipBhattacharyya_12Sept06-1.pdf

Bildsten, L., 1998, *The Many Faces of Neutron Stars*, ed. R. Buccheri *et al*. (Dordrecht: Kluwer), p 419

Bildsten, L., Strohmayer, T., 1999, Physics Today, Feb., p 40

Bildsten, L., and Anthony L. Piro, 2007, *Turbulent Mixing in the Surface Layers of Accreting Neutron Stars*, ApJ 663, p 1252

Brown, B. A., Clement, R. R. C., Schatz, H., & Volya, A., 2002, Phys. Rev. C, 65, 045802

Buttari, A., Amestoy, P., L'Excellent, J.-Y., Guermouche, A., Uçar B., November 23, 2010, *MUMPS: a Multifrontal Massively Parallel Solver*, LyonGrenoble, Toulouse, Bordeaux Workshop CIRA "Systèmes Linéeaires"

Duff, I. S., and Reid, J. K., 1983, *The Multifrontal Solution of Indefinite Sparse Symmetric Linear*, ACM Transactions on Mathematical Software (TOMS) TOMS Homepage archive Volume 9 Issue 3, p 302

Fisker, J. L., Görres, J., Wiescher, M., & Davids, B. 2006, *The Importance of $^{15}O(\alpha,\gamma)^{19}Ne$ to X-Ray Bursts and Superbursts*, ApJ 650, 332

Flynn, M., 1972, *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput. C-21, p 948

Foster, I., 1995, *Designing and Building Parallel Programs*, http://www.mcs.anl.gov/~itf/dbpp/

Fox, G., 1991, *Parallel Problem Architectures and Their Implications for Portable Parallel Software Systems* Tech. Report CRPC-TR91120, Center for Research on Parallel Computation, Rice Univ., Houston, Texas

Fox, J.M., 2007, *Fully-Kinetic PIC Simulations for Hall-Effect Thrusters*. Thesis (S.M.)--Massachusetts Institute of Technology, Computation for Design and Optimization Program, 2007

Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., and Snir, M., 1983, *The NYU ultracomputer: Designing a MIMD, shared memory parallel computer.* IEEE Trans. Computs., C-32(2), p 175

Grindlay, J., Gursky, H., Schnopper, H., Parsignault, D. R., Heise, J., Brinkman, C., & Schrijver, J., 1976, ApJ 205, L127

Gropp W., Lusk, E., and Skjellum, A., 1995, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press

Gupta, A., 2002 *Recent Advances in Direct Methods for Solving Unsymmetric Sparse Systems of Linear Equations*, ACM Transactions on Mathematical Software 28, No. 3, September 2002, p 301

Gupta, A., Joshi, M., Kumar, V., 2001, *WSMP: A High-Performance Shared- and Distributed-Memory Parallel Sparse Linear Equation Solver*, RC 22038 (98932) Computer Science/Mathematics. IBM Research Report

José, J., 1996, *Ph.D. thesis*, University of Barcelona

José, J. & Hernanz, M., 1998, ApJ 494, p 680

José, J. & Iliadis, C., 2011, *Nuclear astrophysics: the unfinished quest for the origin of the elements,* Rep. Prog. Phys. 74

Joss, P. C., 1977, Nature 270, p 310

Hanawa, T., Sugimoto, D., & Hashimoto, M. A., 1983, PASJ 35, p 491

Henyey, L. G.; Forbes, J. E.; Gould, N. L., 1964 *A New Method of Automatic Computation of Stellar Evolution,* ApJ 139, p 306

Koelbel, C., Loveman, D., Schreiber, R., Steele, G., and Zosel, M., 1994, *The High Performance Fortran Handbook*, MIT Press

Koike, O., Hashimoto, M., Arai, K., & Wanajo, S., 1999, A&A 342, p 464

Koike, O., Hashimoto, M., Kuromizu, R., & Fujimoto, S., 2004, ApJ 603, p 242

Kutter, G. S. & Sparks, W. M., 1972, ApJ 175, p 407

Iliadis, C., 2007, *Nuclear Physics of Stars*, Wiley-VCH (eds.)

in 't Zand, J. J. M., Keek, A., Heger, A., Cumming, A., & Weinberg, N., 2009, *Defining the Neutron Star Crust: X-ray Bursts, Superbursts and Giant Flares*, Santa Fe

Intel® Software Development Products, 2011, *Quick-Reference Guide to Optimization with Intel® Compilers*. http://software.intel.com/sites/products/collateral/hpc/compilers/

Kippenhahn, R, Weigert, A, 1996, *Stellar Structure and Evolution*, ed. Springer, p 36

Lafferty, Edward L., 1993, *Parallel computing: an introduction*, ed. NOYD

Lattimer , J., M.,Prakash, M., 2004, *The Physics of Neutron Stars*, Science 304, 536

Lewin, W. H. G., Paradijs, J. V., Taam, R. E., 1993, *X-Ray Bursts*, Space Sci. Rev. 62, p 223

Li, X., March 2010, *Direct Solvers for Sparse Matrices*, Survey article for sparse direct solvers, (http://crd.lbl.gov/~xiaoye/SuperLU/)

Liu, Q. Z., van Paradijs, J., & van den Heuvel, E. P. J., 2007, A&A 469, p 807

Mahaffy, J., 1997, *Introduction to Fortran's Intrinsic Functions*, CMPSC 201, Programming for Engineers, Course notes

Maraschi, L., & Cavaliere, A. 1977, in Highlights in Astronomy, Vol. 4, ed. E. A. Müller (Dordrecht: Reidel), p 127

May, M. M. & White, R. H., 1967, *Hydrodynamic Calculations of General Relativistic Collapse*, p 96

McGinn, S. F., and Shaw, R. E., 2002, *Parallel Gaussian Elimination Using OpenMP and MPI Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'02)*

McGraw, J.R., and Axelrod, T.S., 1988, *Exploiting Multiprocessors: Issues and Options,* in Programming Parallel Processors, R.G. Babb, ed. Addison-Wesley, Reading, Mass., p 7

McKenney, P. E., 2010, *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Linux Technology Center, IBM Beaverton

MPI Forum, 2009, *MPI: A Message-Passing Interface Standard*, Version 2.2

MPICH2, 2011, *Implementation of the Message Passing Interface (MPI) standard* http://www.mcs.anl.gov/research/projects/mpich2/

Moreno, F., 2009, *Ph.D. thesis Accretion onto Neutron Stars: Hydrodynamics and Nucleosynthesis*, UPC

MUMPS, version 4.10.0 (2011) (MUltifrontal Massively Parallel sparse direct Solver), http://graal.ens-lyon.fr/MUMPS/index.php

NASA, 2011, *Imagine the Universe, http://imagine.gsfc.nasa.gov/index.html*

Pacheco, P. S., 1997, *Parallel Programming with MPI*, ed. Morgan Kaufmann

Pancake, C. M. and Cook, C., 1994, *What Users Need in Parallel Tool Support: Survey Results and Analysis,* Proc. Scalable High Performance Computing Conf., IEEE CS Press, Los Alamitos, Calif., p 40

Pancake, C. M., 1996, *Is Parallelism for You?* Oregon State University, IEEE Computational Science & Engineering, summer 1996, p 18

Parikh, A., Jose, J., Moreno, F., and Iliadis, C., 2008, *The effects of variations in nuclear processes on type I X-ray burst nucleosynthesis*, ApJS 178, p 110

Prantzos, N., Arnould, M., & Arcoragi, J. P., 1987, *Neutron capture nucleosynthesis during core Helium burning in massive stars*, ApJ 315, p 209

Rao, V.N. and Kumar, V., 1998, *Superlinear speedup in parallel state-space search*, Foundations of Software Technology and Theoretical Computer Science (Springer, 1988)

Shapiro, S. L. and Teukolsky, S. A., 1983, *Black Holes, White Dwarfs, and Neutron Stars. The Physics of Compact Objects*, Wiley-Interscience, New York

Schatz, H., Bildsten, L., Cumming, A., & Wiescher, M., 1999, ApJ 524, p 1014

Schatz, H., *et al.*, 2001a, Phys. Rev. Lett. 86, 3471
2001b, Nuclear Physics A 688, 150c

Snyder, L., 1986, *Type architectures, shared memory, and the corollary of modest potential* Ann. Rev. Comput. Sci. 1, p 289

Sutter, H., 2008, *Break Amdahl's Law!* (DDJ, February 2008)

Taam, R. E., 1980, ApJ 241, p 358

Taam, R. E., Woosley, S. E., Weaver, T. A., & Lamb, D. Q., 1993, ApJ 413, p 324

Taam, R. E., Woosley, S. E., & Lamb, D. Q., 1996, ApJ 459, p 271

Tapia, R., Lanius, C, 2001, *Computational Science: Tools for a Changing World*, http://ceee.rice.edu/Books/CS/index.html

Thakur, R. and Gropp, W., November 2002, *Improving the Performance of MPI Collective Communication on Switched Networks*, Preprint ANL/MCS-P1007-1102

Venkatesh, T. N., Sarasamma, V. R., Rajalakshmy S., Kirti Chandra Sahu, and Rama Govindarajan, 2005, *Super-linear speed-up of a parallel multigrid Navier–Stokes solver on Flosolver*, Current Science 88, no. 4

Wagoner, R. W., 1969, Synthesis of the elements within objects exploding from very high temperatures ApJS 18, p 247

Wallace, R. K. & Woosley, S. E., 1981, ApJS 45, p 389

Wallace, R. K. & Woosley, S. E., 1984, S. E. Woosley (ed.), New York: AIP

Woosley, S. E. & Taam, R. E., 1976, Nature 263, p 101

Woosley S. E., Heger A., Cumming A., Hoffman R. D., Pruet J., Rauscher T., Fisker J. L., Schatz H., Brown B. A., and Wiescher M., 2004, *Models For Type I X-Ray Bursts With Improved Nuclear Physics,* ApJS 151, p 75

Woosley, S. E., & Weaver, T. A., 1984, *High Energy Transients in Astrophysics*, AIP Conf. Proc. 115, ed. S. Woosley (New York: AIP), p 273

Zima, H., and Chapman, B., 1991, *Supercompilers for Parallel and Vector Computers,* ed. Addison-Wesley